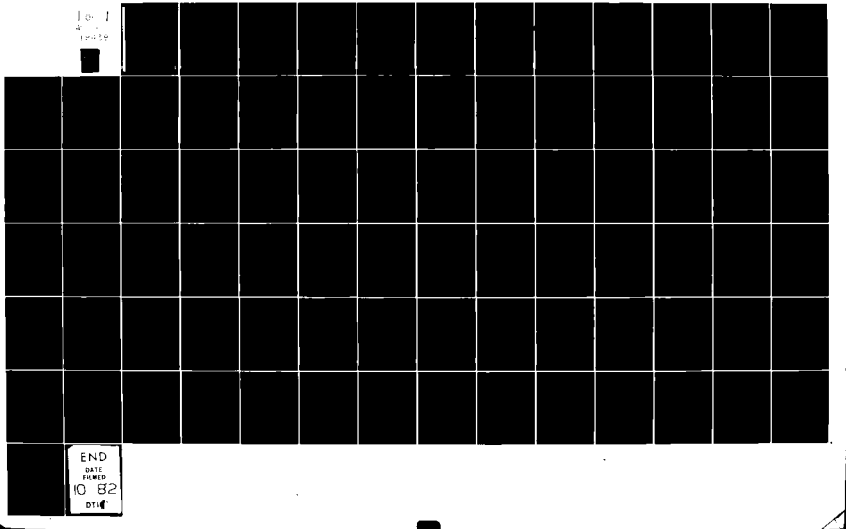
AD-A119 438 STANFORD UNIV CA DEPT OF COMPUTER SCIENCE F/G 12/1
COMBINATORIAL ALGORITHMS I.(U)
MAY 82 E W MAYR
UNCLASSIFIED STAN-CS-82-907 N00014-81-K-0330
NL



END
DATE FILMED
10 82
DTIC

May 1982

(23)

Report. No. STAN-CS-82-907

Combinatorial Algorithms I

by

Ernst W. Mayr

Department of Computer Science

Stanford University
Stanford, CA 94305

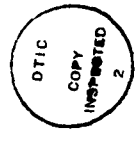
Stanford University
Department of Computer Science

Combinatorial Algorithms I

by

Ernst W. Mayr

May 1982



This report is an edited collection of the class notes prepared for Course 253 A of the same title. The course was taught at the Department of Computer Science at Stanford University in the Winter Quarter 1982. The author wishes to thank the students who took part in that course for their interest which in turn provided much of the motivation to prepare this report, and in particular Tom Spencer, who acted as TA for the course and who helped a lot to make these notes look the way they do.

The publication of this report was supported in part by National Science Foundation grant MCS-77-23738 and by Office of Naval Research contract N00014-81-K-0330.

TABLE OF CONTENTS

Introduction

1.1. Combinatorial Algorithms.

The title of this course is "Combinatorial Algorithms". In order to get some idea of what this means, let's look at the two words in turn (and remember this is not supposed to give a formal, mathematical definition):

- to explain "algorithm" it should actually suffice for all our present purposes if we take it as referring to a "correct computer program (in, say, Pascal or Lisp) guaranteed to terminate on all inputs", or, to be less idealistic, on all *allowed* inputs (but then of course the problem arises which inputs are allowed, and how do we distinguish them!).
- the semantics of "combinatorial" is harder to describe. Classical combinatorics is the science of the properties of *finite* collections of *discrete* objects. Of course, the objects might be anything (like superdenumerable ordinals), but from a combinatiorial point of view we are only interested in certain discrete and finitarily represented properties.

If that is all, you might correctly remark that the word "combinatorial" in "combinatorial algorithms" is simply redundant because (digital) computers on which algorithms supposedly are executed are definitely finitary and discrete.

Hence there must be a more traditional meaning which distinguishes combinatorial algorithms from algorithms in general. Let me try to put this necessarily vague notion perhaps as follows:

> "Combinatorial algorithms are those dealing with problems which would be more or less trivial if one could inspect all possible combinations of the (discrete) objects of any given problem instance."

Of course, this is a very indirect explanation of what a combinatorial algorithm might be, but it has one definite merit to it: it highlights the fact that for all the problems we shall be concerned with we will have to search for solutions other and better than enumerative ones.

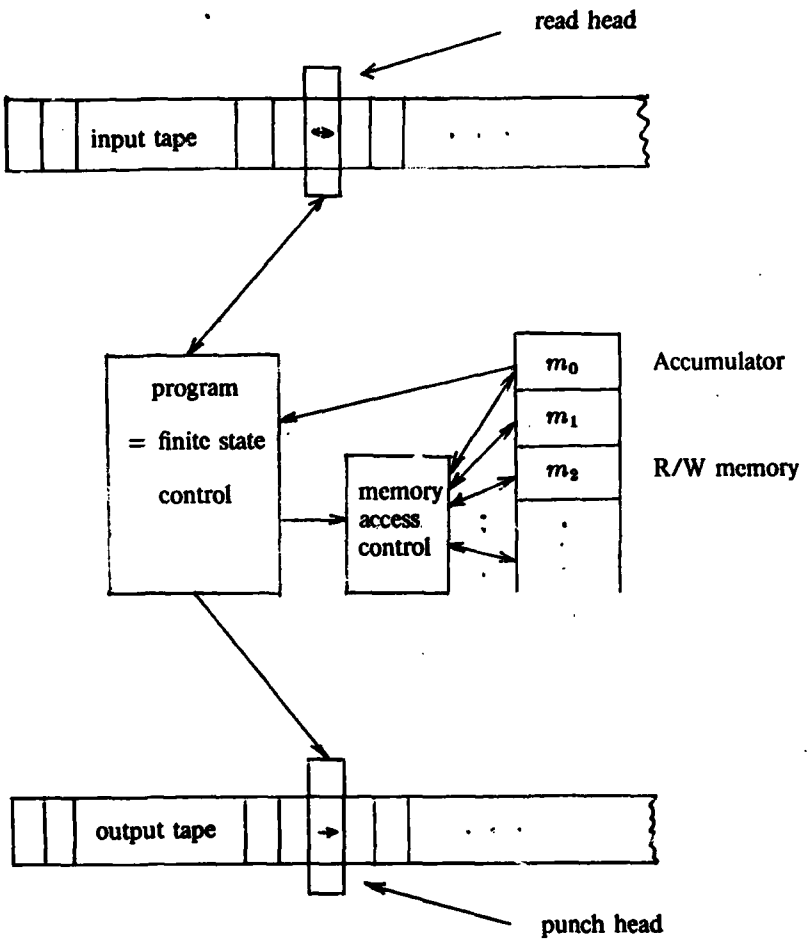
1.2. Machine Models.

We now want to take a somewhat closer look at the computer model which we shall have in mind in most of the cases, or at least in the back of our mind. The reason is that if we intend to make formal (mathematical) statements about computers and the programs running on them we also have to give formal definitions of what computers and programs are. For our purposes, however, it suffices to obtain some rough idea, knowing that when needed there would be a whole elaborated theory to rely on and supply all the missing details.

Thus, our machine model which is called a *Random Access Machine (RAM)*, could be considered as abstracted from an (almost real) computer built, say, at the beginning of the sixties: input and output

would be solely via punched paper tape, program storage would be read-only (quite modern again), and there would be just one register serving as accumulator.

A more formal diagram would look like



— Diagram of RAM —

We idealistically assume that there are infinitely many memory cells $m_0, m_1, \ldots$, and that each of them can hold an arbitrary (signed) integer. (It seems necessary to remark here that more and more often these two assumptions do appear as idealistic; certainly it is possible to discard the second assumption (as even the definitions of some new programming languages do) and simulate arbitrary size integers without (at least theoretically) undue loss of efficiency, but also disposing of the first would just leave us with finite automata). The program or finite state control of a RAM very much looks like assembly language with one address instructions allowing:

- direct, immediate, and indirect addressing for memory transfer and the basic arithmetic and logical operations,
- conditional and unconditional branching,

- input and output, and one or a few control statements (like STOP).

We do not specify any more details because we expect that we will never have to use this rather awkward machine language, and assume that instead have a very sophisticated compiler from a high level *Algol60*- or *Pascal*-like language which even allows us some statements in natural English. This will make the presentation of algorithms closer to our way of thinking and (hopefully) easier to understand. But when analyzing the execution of such programs on a RAM we always have to think of the compiled version as being executed.

1.3. Complexity Measures.

Now that we have presented the basic features of our formal model we want to see how to use it. Well, we would like to employ algorithms and computers in order to get solutions for problems in which we are interested and from which we expect a profit in some sense. Pursuing this economical setup a little bit further, computers are scarce resources, and we incur some costs using them. And, naturally, we are interested in minimizing these costs.

As in economics, we have to clarify two questions first:

(1) how is the cost of running a specific algorithm on a specific (model of a) computer defined, i.e., how do we measure cost?

(2) given two solutions to a problem, how do we compare their respective costs?

1.3.1. Complexity Functions.

Most of the algorithms we shall be looking at are able to solve not only one instance of a problem (say, determining whether the 10,000st digit of the decimal expansion of π is 5), but normally (at least on our idealized computer model, the RAM) an infinite number of such instances (in the above example: finding out whether the n-th digit of the decimal expansion of π is 5, for any positive integer n given to the algorithm). And actually it is just such an infinite collection of instances (given by means of some general parameterization of by a defining common property) what we mean by a *problem*.

In order to be able to work with our formal computer model we also have to find a formal definition of a problem. Here, an instance of a problem would be given by some string, i.e., finite sequence of characters over some fixed, finite alphabet "known" to the computer, which for us somehow describes the problem instance we have in mind. As the execution of an algorithm, given some string as input, is independent of what meaning we attach to the string, we arrive at the following formal

Definition:

(a) A *problem* is a subset L of the set Σ^* of strings over some (fixed, finite) alphabet Σ.

(b) An algorithm A (on a RAM) *recognizes* the problem L if, for any $x \in \Sigma^*$, on input x [†] the execution of A eventually stops and the output is

"yes", if $x \in L$,
"no", otherwise.

(c) We say that A *accepts* L if, for any $x \in \Sigma^*$, on input x the execution of A eventually stops with output "yes" if and only if $x \in L$.

(So we do not care what A does on any input $y \notin L$ except that it must not stop with output "yes".)

Returning to our trivial example above, we might not be interested only whether the n-th digit of π is 5, but instead would like, on input n, to obtain the first n decimal digits of π. Here we are not dealing with a *recognition problem* with a simple yes/no answer, but with the more general problem of *computing a function*, say from Σ^* to Σ^*:

[†] We assume by convention that Σ is such that A is able to determine the end of input on its input tape.

Definition:

Let $f : \Sigma^* \to \Sigma^*$ be a (in general partial) function. An algorithm A (on a RAM) *computes* f if, for any $x \in \Sigma^*$, on input x the execution of A eventually stops if and only if $f(x)$ is defined if and only if A stops with output $f(x)$.

For a given problem L or function f, there naturally are — as in real world economics — many different "costs" or, as we call them, *complexities*, and it always depends on the circumstances which ones are relevant or interesting. Some are *static* in the sense that they do not depend on the problem instances, e.g.

- the minimal size (= number of instructions) of an algorithm which recognizes L or computes f;
- the minimal number of branching instructions in an algorithm for L or f.

Other complexity measures are *dynamic* in that they do depend on problem instances, i.e., the input. The two most important kinds of dependencies are

- *worst-case complexity*, where, for any $n \in \mathbb{N}$, the maximum cost for any x of length n is taken with
  (a) $x \in \Sigma^*$ for a recognition problem,
  (b) $x \in L$ for an acceptance problem, and
  (c) $x \in \mathrm{dom} f$ for computing a function f.
- *expected* or *average complexity*, where, for some 'size' function and probability distributions over all inputs of the same 'size', the average cost (according to the corresponding probability distribution) over all inputs of each 'size' is taken.

We shall mostly emphasize worst-case complexity because it is universal (with respect to the underlying model) and does not depend on the choice of more or less arbitrary probability distributions.

The two most important types of costs or complexity which we shall be considering are those concerned with (computation) *time* and (storage) *space*.

The *time* used in the execution of an algorithm A on some input $x \in \Sigma^*$ is the sum of the time spans for all executions of the basic instructions in the algorithm.

The *space* used in the execution of A on input x is the sum of the 'spaces' occupied by the memory cells referenced during the execution of A.

To complete the definition of time and space complexity of an algorithm, it remains to state the time required to execute each RAM instruction and the space used by a memory cell. Again there are two commonly used ways to define these costs. The first is called the *uniform cost criterion*. It attributes one unit of cost (time or space) to each execution of a basic RAM instruction and to each memory cell, regardless of its contents. Unless otherwise mentioned we shall refer to this cost criterion. It is appropriate as long as a real world computer can simulate the RAM algorithm using only some fixed amount of time and storage words per basic RAM instruction and memory cell. In principle this simply means that the numbers stored in the RAM memory cells must not become too big. If this should happen, a somewhat more realistic approach might be the so-called *logarithmic cost criterion* which defines the space used by a memory cell to be the length of the binary representation of the biggest (in absolute value) number stored in the memory cell during the execution of the algorithm, and which assigns to each basic instruction the sum of these lengths of all its operands as the time span used. A detailed example of logarithmic costs for a RAM instruction set can be found in [AHU74]. We remark that the time and/or space complexity of some algorithm may be widely different depending on whether the uniform or logarithmic cost criterion is employed.

1.3.2. Asymptotic Complexity.

We now address the problem of comparing complexity functions. Assume that for a certain problem we have two algorithms, one with complexity $f(n) = n^{2.81} \log n$, the other with complexity $g(n) = n^3 / \log n$ [†]. Though for small n the second algorithm has a smaller complexity, there is a break point

[†] $\log n$ denotes the real-valued solution to $2^x = n$.

n_0 ($n_0 \approx 39$) such that for $n \geq n_0$ the situation is reversed (if the complexity considered is worst-case complexity this does *not* mean that for all inputs of size at least n_0 the first algorithm has a lower complexity that the second; there must be however infinitely many such instances).

Most of the times we are interested in comparing two algorithms when applied to problem instances of ever bigger size. The complexity of an algorithm for the input size tending to infinity is called its *asymptotic complexity*, and using it as criterion we might therefore say that the first algorithm above has a lower complexity than the second (note however that in practical life the situation is not as simple, and that for a given problem instance the selection of the better algorithm always depends on the break point).

For the comparison of complexity functions in the asymptotic sense, we use the following notation:

Definition:

Let $f, g : \mathbf{N} \to \mathbf{N}$ be two functions; we say that

(a) $f = O(g)$ if $\exists c > 0$ such that $f \leq cg$;

(b) $f = o(g)$ if $\forall c > 0 \quad f \leq_{a.e.} cg$
($f \leq_{a.e.} g$ means $f(n) \leq g(n)$ for all but finitely many n);

(c) $f = \Omega(g)$ if $\exists c > 0$ such that $f \geq_{a.e.} cg$;

(d) $f = \Theta(g)$ if $f = O(g)$ and $f = \Omega(g)$.

(See [Knu76].)

1.4. Reduction and Recurrences.

When designing an algorithm for some problem it often happens that we can derive a solution for any instance with, say, relatively little cost if we only have the solutions to a few other, smaller instances of the same problem or of instances of some other problem which we know how to solve. In the second case it is immediate to determine the complexity of an algorithm for the first problem which uses this reduction, from the known complexity of the second algorithm and the additional cost for the reduction. When we wish to determine the complexity of an algorithm constructed according to the first case, however, we usually end up with a relation of the following form for the complexity C of our algorithm:

$$C(n) = F_1(C(n^{(1)}, \ldots, C(n^{(i)})) + F_2(n),$$

where F_1 describes the dependency on the complexities of the smaller problem instances of size $n^{(1)}, \ldots, n^{(i)}$, and F_2 represents the additional cost incurred to form a solution for the original problem instance of size n. We shall now discuss some techniques for solving such recurrence relations, i.e., finding closed-form expressions for the function C when the function F_1 is sufficiently simple. Nonetheless, these methods provide a systematic way to deal with a fairly large class of recurrence equations (also see [Luc80]). The techniques are, in turn, *multipliers*, *characteristic polynomials*, *generating functions*, and *domain* and *range transformations*. We shall demonstrate each technique by means of an example where, for notational convenience, we write f_n for $f(n)$.

1.4.1. Multipliers

Suppose that the following recurrence relation is given:

$$\begin{aligned} f_n &= 2f_{n-1} + n; \qquad \text{for } n \geq 2, \\ f_1 &= 1; \end{aligned}$$

That is

$$f_n = 2f_{n-1} + n; \qquad / \cdot 1$$

$$\begin{aligned} f_{n-1} &= 2f_{n-2} + (n-1); && / \cdot 2 \\ f_{n-2} &= 2f_{n-3} + (n-2); && / \cdot 4 \\ &\vdots \\ f_{n-i} &= 2f_{n-i-1} + (n-i); && / \cdot 2^i \\ &\vdots \\ f_2 &= 2f_1 + 2; && / \cdot 2^{n-2} \end{aligned}$$

If we multiply each equation with the multiplier indicated to its right, then sum up all equations and cancel identical terms on both sides, we obtain

$$\begin{aligned} f_n &= 2^{n-1} f_1 + \sum_{i=0}^{n-2} (n-i)2^i \\ &= \sum_{i=0}^{n} (n-i)2^i \qquad \text{because } f_1 = 1 \\ &= \sum_{i=0}^{n-1} \sum_{j=0}^{i} 2^j = 2^{n+1} - n - 2. \end{aligned}$$

The last line follows by elementary transformations (also see 1.4.3.). Hence we obtain

$$f_n = 2^{n+1} - n - 2.$$

1.4.2. Characteristic Polynomials.

As an example, suppose we are given

$$\begin{aligned} f_n &= f_{n-1} + f_{n-2}; \qquad \text{for } n \geq 2 \\ f_1 &= 1, \quad f_0 = 0. \end{aligned}$$

Such a recurrence is called a *homogeneous linear recurrence with constant coefficients.*

If we try to set $f_n = a^n$ for some a still to be determined we get the following *characteristic equation* for a (assuming that $a \neq 0$):

$$a^2 - a - 1 = 0, \text{ and hence } a_{1,2} = \frac{1 \pm \sqrt{5}}{2}.$$

It is clear that every linear combination

$$c_1 a_1^n + c_2 a_2^n$$

is then a solution to the recurrence relation $f_n - f_{n-1} - f_{n-2} = 0$ for $n \geq 2$. The boundary conditions for f_0 and f_1 determine specific values for c_1 and c_2, in our example $c_1 = 1/\sqrt{5} = -c_2$, and hence

$$f_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right].$$

The question is whether by this method we can obtain all solutions to recurrences like the one above. It is not hard to see that this is true if all roots of the characteristic polynomial are distinct. For the general case, with multiple roots allowed, we cite the following theorem (for a proof see [Liu68]):

Theorem:

Let $p(x)$ be the characteristic polynomial of the recurrence

(*) $$p_0 f_n + p_1 f_{n-1} + \ldots + p_k f_{n-k} = 0; \qquad \text{for } n \geq k$$

with constant coefficients p_i. Let the roots of p, over the complex numbers, be r_i, $i = 1, \ldots, m$, each with multiplicity q_i. Then all solutions of (*) are given by

$$f_n = \sum_{i=1}^{m} (r_i^n \sum_{j=0}^{q_i - 1} c_{ij} n^j),$$

where the c_{ij} are arbitrary constants.

If the given linear recurrence relation (with constant coefficients) is *inhomogene* ... first try to derive a homogeneous recurrence using appropriate multipliers, and then apply the ... hod discussed above. A more formal treatment can be found in [Lue80].

1.4.3. Generating Functions.

For a sequence $(f_n)_{n \geq 0}$, its *generating function* $F(z)$ is defined as

$$F(z) = \sum_{n \geq 0} f_n z^n.$$

Just using calculus, we can establish the following table:

generating function	n-th element of the corresponding sequence
cF	cf_n
$F + G$	$f_n + g_n$
$F \cdot G$	$\sum_{i=0}^{n} f_i g_{n-i}$
$z^k F$	if $n < k$ then 0 else f_{n-k}
$F(z)/(1-z)$	$\sum_{i=0}^{n} f_i$
$z \frac{dF}{dz}(z)$	nf_n
$\int_0^z F(t)dt$	if $n = 0$ then 0 else f_{n-1}/n
$F(cz)$	$c^n f_n$

Hence, if we put

$$F(z) = \sum_{n \geq 0} 2^n z^n = \frac{1}{1-2z};$$
$$G(z) = \sum_{n \geq 0} n z^n = \frac{z}{(1-z)^2},$$

we obtain

$$F(z) \cdot G(z) = \frac{z}{(1-z)^2(1-2z)} = \sum_{n \geq 0} \sum_{i=0}^{n} (n-i) 2^i z^n.$$

Decomposing $z/((1-z)^2(1-2z))$ into partial fractions we get

$$\frac{z}{(1-z)^2(1-2z)} = \frac{-2}{1-z} - \frac{z}{(1-z)^2} + \frac{2}{1-2z},$$

and thus the table above provides the answer

$$\sum_{i=0}^{n}(n-i)2^i = 2^{n+1} - n - 2.$$

1.4.4. Domain and Range Transformations.

A real-valued sequence $(f_n)_{n\geq 0}$ is a mapping from N to the reals. Thus a transformation on the values of the sequence is called a *range transformation* and a transformation on the indices a *domain transformation.* We first show an example of a range transformation. Suppose we are to solve the recurrence

$$\begin{aligned} f_n &= f_{n-1} \cdot f_{n-2}, \qquad \text{for } n \geq 2, \\ f_1 &= 2, \quad f_0 = 1. \end{aligned}$$

If we let

$$g_n = \log f_n,$$

we may rewrite the recurrence as

$$\begin{aligned} g_n &= g_{n-1} + g_{n-2}, \qquad \text{for } n \geq 2, \\ g_1 &= 1, \quad g_0 = 0. \end{aligned}$$

For this recurrence, we know the solution (see 1.4.2.; g_n is the n-th *Fibonacci* number F_n), and substituting back yields

$$f_n = 2^{F_n}.$$

A domain transformation is conveniently used in the following example. Let

$$\begin{aligned} f_n &= 3f_{n/2} + n, \qquad \text{for } n = 2^k > 1, \\ f_1 &= 1. \end{aligned}$$

Here we substitute the index and define

$$g_k = f_{2^k},$$

from which we get the following secondary recurrence for the g_k:

$$\begin{aligned} g_k &= 3g_{k-1} + 2^k, \qquad \text{for } k \geq 1, \\ g_0 &= 1, \end{aligned}$$

which can be solved with methods discussed earlier. We obtain $g_k = 3^{k+1} - 2^{k+1}$, and therefore

$$f_n = 3n^{\log 3} - 2n.$$

This concludes our discussion of several very useful techniques to solve recurrence relations for sequences. Additional material can be found for instance in [Mil60,Rei77,Rio58,Rio68].

Higher Level Data Structures

2.1. Basic Set Operations.

It is often useful to design algorithms in layers. That is, first the algorithm is described in a high level way with abstract structures that are described in a fashion that is independent of their implementation. Then the implementation details of the structures are worked out or discovered in the literature.

It is necessary to talk about the specifications rigorously. To do this, the language of set theory is used. There are several operations that are often performed on data structures. These involve $a, a_1, a_2, \ldots$ as data elements and $S, S_1, S_2, \ldots$ as sets of data elements. The operations are:

1. Member (a, S): is $a \in S$?
2. Insert (a, S): $S \leftarrow S \cup a$.
3. Delete (a, S): $S \leftarrow S - a$.
4. Replace (a, a', S): $S \leftarrow (S - a) \cup a'$.
5. Union (S_i, S_j): $S_i \leftarrow S_j \cup S_i$.
6. Find (a): if $a \in \cup S_i$ then i such that $a \in S_i$ otherwise undefined.

If in addition there is a totally ordered universal set which contains all possible data elements, the following operations make sense.

7. Min (S): if $S = \emptyset$ then undefined else $\min\{b;\ b \in S\}$.
8. Max (S): if $S = \emptyset$ then undefined else $\max\{b;\ b \in S\}$.
9. Split (S_i, a, S_j): if $a \in S_i$ then $S_j \leftarrow \{b \in S_i; b > a\}$; $S_i \leftarrow \{b \in S_i; b \leq a\}$, otherwise undefined.
10. Concatenate (S_i, S_j): if $\max(S_i) < \min(S_j)$ then $S_i \leftarrow S_i \cup S_j$ else undefined.

It is ususally necessary to be able to perform several of these operations on the same sets. Some of these combinations of operations have names and well know implementations.

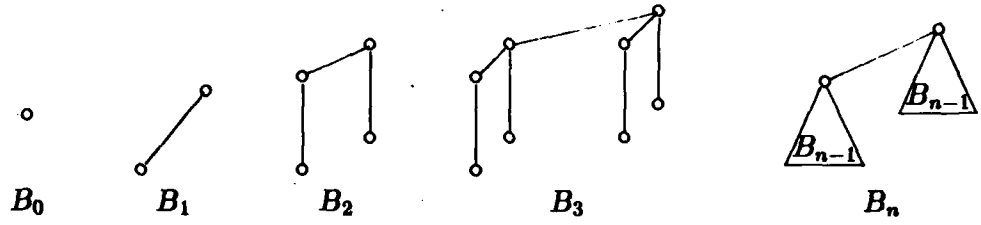
name	supported operations	implementations
dictionary	member, insert, delete	hash table, balanced trees
priority queue	insert, delete, min	balanced, leftist trees, heap, binomial forest
mergeable heap	insert, delete, min, union	2-3 tree, binomial forest, leftist tree
concatenable queue	insert, delete, split, concatenate	2-3 tree

Here "balanced tree" means any of several balanced tree schemes, including 2-3 trees, AVL trees, and RB-trees.

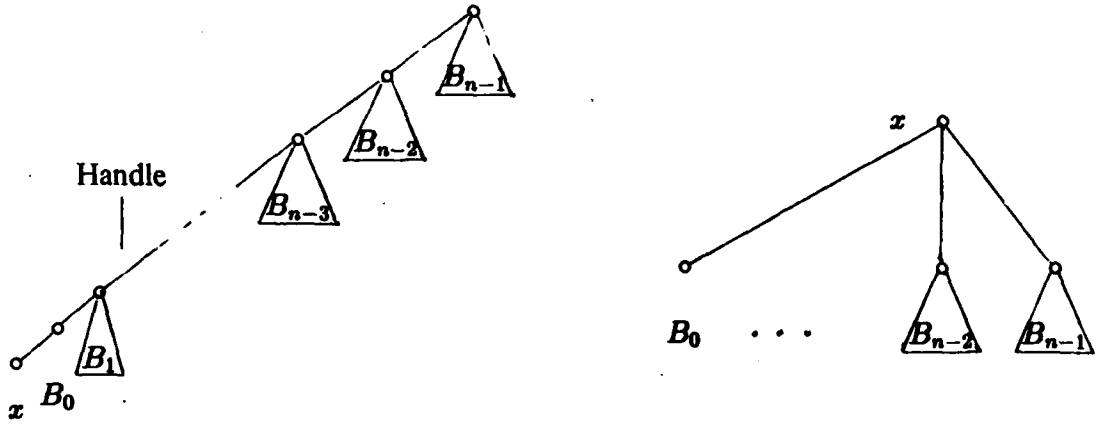
2.2. Binomial Queues.

2.2.1. Definitions.

Binomial queues are used to implement priority queues and mergeable heaps. These queues are based on binomial trees. Binomial trees are defined recursively. The smallest one, B_0 is a single node. In general, B_n consists of two B_{n-1}'s, such that the root of one is made the son of the root of the other. A picture should help.



There are two ways of thinking of binomial trees, depending on whether the sons of the root or the longest path from the root to a leaf is being concentrated on. This path is called the handle. There is one tree of each size on the handle. Again a picture should help.



A binomial tree has the following properties:

$$|B_n| = 2^n$$

$$height(B_n) = n$$

$$\text{there are } \binom{n}{k} \text{ nodes at depth } k$$

We have seen how to have a number of nodes in a tree that is a power of 2. It is necessary to allow structures with an arbitrary number of nodes, n. Consider the binary expansion of n, $n = \sum b_i 2^i$. Then the *binomial forest*, F_n, of n nodes, includes a B_i if and only if $b_i = 1$.

A binomial queue is a binomial forest with a key, K_i attached to every node, i, so that the "heap condition" is satisfied. The "heap condition" states that if node j is a descendent of node i then $K_i \leq K_j$.

2.2.2. Union.

Now it is necessary to consider how the operations are implemented. First consider the operation of union. That is, there are two forests, F_i and F_j that need to be merged. There are two cases to consider:

a) If i and j are powers of 2 then if $i \neq j$ the result is simply the two trees. Otherwise, the two trees must be joined. This is done by having the root with the smaller key become the father of the other root.

b) In general, a process like that of binary addition is needed. Each size of tree from smallest to largest is considered. Sometimes there is a side tree called the carry present. Initially it is absent. For any size there are from zero to three trees present, possibly a carry and possibly one tree from each forest. If there are zero trees present then there is no tree of that size in the result and no carry for the next stage. If there is exactly one tree present, that tree is present in the result and there is no carry. If two trees are present then they are merged as in case (a) to form the carry. There is no tree of that size present in the answer. If three trees are present, then two are merged to form the carry and the third is present in the answer. The cost of this operation is bounded by the number of trees in the forests. Therefore the cost of union(F_i, F_j) is $O(\log(i) + \log(j))$.

2.2.3. Insertion.

The operation of insertion is conceptually the same as union with a one element forest. There are, however, subtleties to the analysis of the cost that make it worth while to be considered separately. Since one of the forests consists of one node, once the chain of carries stops, the rest of the remaining forest can be added in constant time. Therefore, the cost of adding n nodes to a binomial queue is $O(n + c)$, where c is the number of trees merged. Each edge in the forest represents the fact that two trees have been merged. The number of edges in F_m is $m - w(m)$, where $w(m)$ is the number of 1's in the binary representation of m and the number of trees in F_m. Therefore $c = (m+n-w(m+n))-(m-w(m)) = n + (w(m+n) - w(m))$. Since $0 \leq w(j) \leq log(j)$, $c \leq n + \log(n+m)$. Therefore, the cost of doing n insertions into an m node forest is $O(n + \log(n+m))$

2.2.4. Min.

To find the smallest element in a forest, it is necessary to look at the root of each tree to find the smallest. There are $w(n) = O(\log(n))$ of these.

2.2.5. Deletion.

The deletion operation is the most complicated. The node that is to be deleted is in some tree, B_i. The deletion operation consists of somehow decomposing this tree into a forest consisting of the node to be deleted and one tree of each smaller size. This forest is then merged with the original forest with B_i deleted. When considering the decomposition of B_i, there are two cases, either the node to be removed is the root or it is not. If the node is the root, then the second way of looking at binomial trees determines the decomposition. If the node to be deleted is not the root it is a member of some B_j on

the handle. The trees below B_j on the handle form a B_j. That tree and the trees above B_j are part of the decomposition. The trees smaller than B_j in the decomposition are found by recursively deleting the node from B_j. Each tree can be found in unit time. Therefore, the time to delete a node from F_n is $O(\log(n))$. Note that for this to be done the node must first be found. A dictionary is useful for doing this.

The details of how to represent binomial trees are covered in the homework. For an extended discussion, also see [Bro78].

Some applications of binomial queues include scheduling, discrete simulation, sorting, optimal code construction, shortest path algorithms, minimum cost spanning tree algorithms. Binomial trees are important in a linear median algorithm and other places.

Selection — the Median Problem

Suppose we are given a totally ordered universe $(U, \leq)$ and we want to find, for some i with $1 \leq i \leq n$, the i-th largest element of a given finite $S \subseteq U$ with n elements. For $i = 1$, this is the problem of finding the minimum (resp., maximum, if we reverse the order) of a set. It can (and in sports usually is) solved by an *knock out tournament*, and it is not hard to see that this method minimizes the number of comparisons (i.e., the number of "matches"). However, when it comes to determining the vice champion, this need not be, by any means, the loser of the final match. As a matter of fact (though this is not commonly realized in the world of sports), it might be any one of the participants in the tournament who was thrown out by losing against the final winner (of course, we assume here transitivity of the "is a better player than" relation). And it was also this apparent injustice which might have led to the formulation of the first selection algorithm [Car83] (for $i \neq 1, n$).

Another variant of the selection problem which deserves specific mentioning is the so-called *median problem*. It calls for the determination of the $\lceil n/2 \rceil$-th largest element and is of practical importance whenever we want to split a set into two equal sized parts such that the elements in the one part are all smaller and those in the other all bigger than the splitting element (we may assume here without loss of generality that n is odd).

In the sequel, we are going to discuss two algorithms for selection problems. For the analysis of these algorithms, we choose to take the number of comparisons made by the algorithm between elements of the set, as the dominant part of the overall complexity. In these terms, both algorithms turn out to be of linear complexity.

3.1. The Blum-Floyd-Pratt-Rivest-Tarjan Selection Algorithm.

Let m be a small, odd number (say, $5 \leq m \leq 15$), and consider the following algorithm for selecting the i-th smallest element of S for any i between 1 and n.

Linear Selection Algorithm 1:

1. Divide S into $\lceil n/m \rceil$ subsets, $\lfloor n/m \rfloor$ of them with m elements.
2. Sort these $\lfloor n/m \rfloor$ subsets; if $\lfloor n/m \rfloor$ is even, also sort the remaining subset minus one element. Let S' be the set of the medians of these sorted subsets.
3. Determine recursively the $\lceil |S'|/2 \rceil$-th element of S', call it s.
4. Divide S into $S_1 := \{s' \in S; s' < s\}$ and $S_2 := \{s' \in S; s' > s\}$
   (note that $|S_1|, |S_2| \geq n/4$ if we assume, and we do this without loss of generality, that $n \geq 3m - 1$).
5. If $S_1 \geq i$ determine recursively the i-smallest element of S_1;
   if $S_2 > n - i$ recursively determine the $i - |S_1| - 1$-smallest element of S_2;
   otherwise return s.

The worst-case complexity T of this algorithm (measured in number of comparisons) hence is

bounded by

$$T(n) \leq T(\lceil n/m \rceil) + T(\lfloor 3n/4 \rfloor) + \lceil n/m \rceil \cdot C_m + \lfloor n/2 \rfloor + m,$$

where C_m is the number of comparisons needed to sort m elements (using, say, the Ford-Johnson sorting algorithm). By induction we obtain

$$T(n) \leq c_m n \qquad \text{with } c_m \approx 20.$$

For further details see [BFP73].

3.2. The Schönhage-Paterson-Pippenger Median Algorithm.

This algorithm is the best algorithm known so far with respect to its asymptotic complexity (expressed in number of comparisons used by the algorithm). It makes essential use of binomial trees. It is also of interest because it is not recursive (based on a divide-and-conquer approach) but rather is best described as a *pipelined* algorithm.

Definition:

Let S_k denote the partial order consisting of $2k+1$ elements k of which are smaller, and k of which are bigger than the remaining element, which is called the *center* of S_k.

We are now going to define a special variant of binomial trees with one distinguished node, called the *center*. The nodes of the trees are elements from a totally ordered universe.

Definition:

(a) The tree H_0 is a singleton node, which is also its center.
(b) The tree H_1 is obtained by combining two H_0's by an edge; its center is the smaller of the H_0's.
(c) The tree H_2 is obtained by combining two H_1's by an edge; its center is the bigger of the centers of the two H_1's.
(d) The tree H_{2h}, for $h > 0$, is obtained by combining two H_{2h-1}'s by an edge; its center is the smaller of the centers of the two constituents.
(e) The tree H_{2h+1}, for $h > 0$, is obtained by combining two H_{2h}'s by an edge; its center is the bigger of the centers of the two constituent trees.

(Note that there is an irregularity in the definition of the H_i just for small i; $H_0, H_3, H_5, \ldots, H_{2h+1}, \ldots$ are of the same "variety", and so are $H_1, H_2, H_4, \ldots, H_{2h}, \ldots$.)

We now state those properties of the H_h which will be needed in the algorithm, in the following

Decomposition Lemma:

(a) H_h has 2^h nodes, exactly $2^h - 1$ comparisons are needed to produce it (because it is a tree!).
(b) For $h \geq 1$, H_{2h} can be decomposed into
  -its center,
  -a (disjoint) collection $\{H_0, H_3, H_5, \ldots, H_{2h-1}\}$ of subtrees all of whose centers are above the center of H_{2h}, and
  -a collection $\{H_1, H_2, H_4, \ldots, H_{2h-2}\}$ of subtrees all of whose centers are below the center of H_{2h}.
(c) H_r can be split such that the component of the center contains exactly $2^{\lfloor r/2 \rfloor}$ elements all at or
  α) above the center if $r = 0, 3, 5, 7, \ldots$,
  β) below the center if $r = 1, 2, 4, 6, \ldots$,
  by removing at most $\frac{1}{2} \cdot 2^{\lfloor r/2 \rfloor}$ edges.

(d) If $k \leq 2^h - 1$ then H_{2h} can be decomposed such that the component of the center contains $2k+1$ elements, k of which are bigger, and k of which are smaller than the center (i.e., the component contains S_k); for this, at most $3k + 2h$ edges have to be cut, and the remaining components are of the form H_s, $s < 2h$.

The proof of this Decomposition Lemma is left as a homework problem. For further details, also see [SPP76].

Now suppose that we have a chain of t pairwise disjoint S_k's whose nodes are taken from some set with n elements, and let r be the number of leftover elements (i.e., $n = (2k+1)t + r$). Also assume that the centers of the S_k's appear in the chain in increasing order, and let *top* (*bot*) be the last (first) center in the chain. Under the assumption that r is less than $t-1$, we can then conclude that *top* is known to be bigger than

$$k + (t-1)(k+1) > k + (k+1)(\frac{n+1}{2k+2} - 1) = \frac{n-1}{2}.$$

Hence *top* and the elements known to be bigger than *top* are also known to be bigger than the median of the n element set. A corresponding situation applies to *bot*. Thus, $2(k+1)$ elements can be discarded from the set and the median determined as the median of the remainder.

The (basic version of the) SPP Median Algorithm is shown in the diagram on the next page. There, n is the number of elements for which we search the median, k is set to $\lfloor n^{1/4} \rfloor$, and h is chosen such that $2^{h-1} \leq k < 2^h$.

The algorithm maintains two pools, one of which contains H_s's, the other just singletons (the two pools actually could be merged; for subsequent improvements of the basic algorithm, it is however convenient to have them separate). Initially all elements of the set are in this second pool. The first process in the algorithm recursively forms H_{2h}-trees, whenever there is enough building material available in the two pools. This means in particular that, whenever there are two H_s's with the same s in the pools, they are combined to produce an H_{s+1}. Whenever an H_{2h} gets produced by this formation process, it is decomposed into a component containing S_k and various H_s's (which are being recycled in their pool) according to the Decomposition Lemma stated above. The S_k-component is then inserted into a chain of such components in a way such that their centers are in increasing order. Whenever none of the processes discussed so far can run any more, one of two other processes is started, depending on the relation between the length t of the S_k-chain and the number r of elements which are leftover elements outside of the chain. If $t - 1 > r$ then, as argued above, $2k+2$ extremal elements in the chain can be discarded because they cannot possibly be the median, and as half of them lie below and half of them lie above the original median, we are still left with determining the median of the remaining elements. If $t - 1 \leq r$ the following analysis will show that the number of elements remaining in the whole process is small enough so we can sort them and thus determine their median.

Analysis of the SPP Median Algorithm.

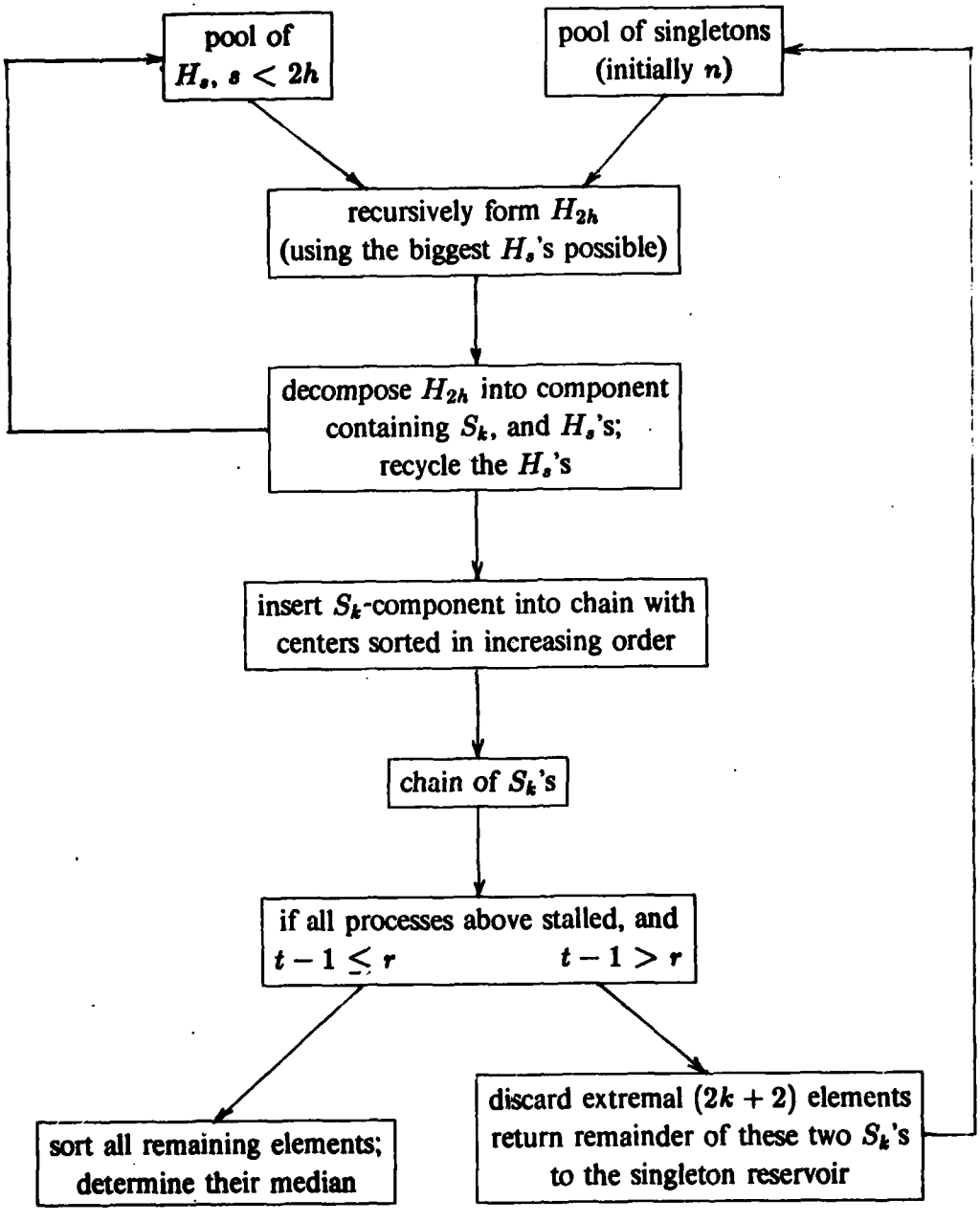
Consider the following quantities:

$r :=$ the number of leftover elements (elements which are stuck in the recursive H_{2h} formation process respectively its input pools). Note that at most one H_s can be left over for every $s < 2h$, and as H_s contains 2^s elements, we have

$$r \leq 2^{2h} - 1.$$

$R :=$ the number of elements to be sorted in the final step of the algorithm; these are the elements in the chain when this last step is invoked, and the leftover elements. As $t \leq r + 1$ in this case, we have

$$R = (2k+1)t + r \leq 2^{2h}(2k+1) + 2^{2h} - 1.$$



— Diagram of the basic SPP Median Algorithm —

$m :=$ the total number of S_k-components produced during the whole process. As, whenever $2k+2$ elements are discarded from the process, two S_k-components are destroyed, m equals twice the number of such discarding actions plus the number t of S_k-components left at the end. Hence,

$$m = 2 \cdot \frac{n-R}{2k+2} + t = \frac{n-R}{k+1} + t.$$

We are now able to count $T(n)$, the total number of comparisons made by the algorithm. As the algorithm always preserves acyclicity of all graphs involved. $T(n)$ is equal to the number of edges generated during the algorithm. And this number in turn is bounded above by

the number of edges in all S_k-components
plus the total number of edges removed in the decomposition process
plus the number of edges left at the end in the pools of the H_{2^h}-formation process,
plus the total number of comparisons for the chain insertion process,
plus the number of comparisons for the final sorting process.

Together this yields

$$T(n) \leq (\frac{n-R}{k+1} + t)[2k + 3k + 2h + \log(n/(2k+1))] + R\log R + r.$$

As $k = \lfloor n^{1/4} \rfloor$, and $2^{h-1} \leq k < 2^h$, we have

$$\begin{aligned} r &= O(k^2); \\ t &= O(k^2); \\ R &= O(k^3), \qquad \text{and hence} \end{aligned}$$

$$T(n) \leq 5n + o(n).$$

Some improvements are possible for this basic version of the algorithm. Their description would be very laborious, and the reader is referred to [SPP76] for the details. Implementing these changes, we obtain a median algorithm whose asymptotic complexity in terms of number of comparisons is $3n + o(n)$, the best known upper bound so far.

Theorem:

The median of a set of n elements can be determined using at most $3n + o(n)$ comparisons.

3.3. A Lower Bound For the Median Problem.

We have seen that it is possible to compute the median of an n-element set in linear time. The question of how many comparisions are needed to do this naturally arises. The best published result of $7/4n - o(n)$ is due to Pratt and F. Yao [PFY73]. Here a weaker result due to Blum, Floyd, Rivest, Pratt, and Tarjan [BFP73] will be discussed.

Theorem:

Finding the median of an n element set requires at least $\lceil 3n/2 \rceil - 2$ comparisons.

Proof: The proof of this theorem uses what is called an adversary argument. An algorithm A asks questions of the form: "is $x \leq y$?" The adversary, B, gives answers so that all of its answers are consistent and A has to ask "many" questions.

It is necessary to describe B in greater detail. Suppose n is odd and $m := \lceil n/2 \rceil$. B maintains three sets of elements: U, L, G. B decides that all of the elements in G are greater than the median and

that all of the elements of L are less than the median. The elements of U are candidates for being the median. B also ensures that if $g \in G$, $l \in L$, and $u \in U$ then $g \geq u \geq l$.

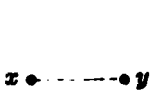
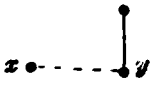
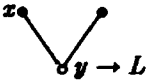
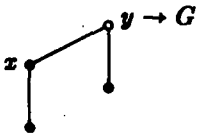
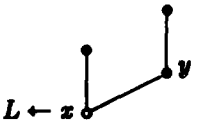
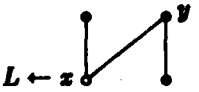
Basic observation:

As long as $|L|, |G| < \lceil n/2 \rceil$ it is consistent for B to assume that the median is in U. If in addition, there are two incomparable elements in U, A has cannot have determined the median yet.

Thus, B will try to keep L and G small and U big. For simplicity, however, B will keep in U only those elements which have been compared to at most one other element currently in U. If this condition is violated due to a question from A, appropriate elements are removed from U and put into L or G so as to restore this condition. B always answers such questions so that there are never three elements of U, a, b, c such that a is known to be less than b and b is known to be less than c.

First phase of the adversary algorithm.

Initially, $L = G = \emptyset$, and U contains all n elements. If A asks a question "Is $x \leq y$?", and x and y are not both in U B answers it so that it is consistent and the invariance $L"\leq", \ldots, U, \ldots, "\leq", \ldots, G$ is maintained. From B's point of view this is a redundant comparison. If x and y are both in U then B answers the question depending on which of x and y, if any, have been compared and what the result was. The six cases are drawn below. A line between two elements indicates that the higher one is known by A to be greater than the lower one. A dotted line represents the question A is asking.

	state of x, y	B's action	number of pairs	$\lvert U\rvert$	$\lvert L\rvert$	$\lvert G\rvert$
1)			+1	—	—	—
2)			−1	−1	—	+1
3)			−1	−1	+1	—
4)			−1	−1	—	+1
5)			−1	−1	+1	—
6)			−1	−1	+1	—

If C is the number of comparisons made so far, and P is the number of pairs in U, then it is obvious by induction that :

$$C - P + 2|U| \geq 2n.$$

Second phase of the adversary algorithm.

The first phase of the adversary algorithm stops if either $|L| = \lfloor n/2 \rfloor$, $|G| = \lfloor n/2 \rfloor$, or there are not two incomparable elements in U. Suppose the latter happens. Then $|U| \leq 2$, so $C - P + 2(2) \geq 2n$. This means that $C \geq 2n - 3$.

If, on the other hand, L or G gets too large, B enters a second phase. Suppose without loss of generality, that it was L that became to big. B will then force A to find the minimum element of U, which will be the median. This phase will require an additional $|U| - P - 1$ comparisons, because $|U| - 1$ comparisions are necessary to find the minimum of $|U|$ elements and at most P of the comparisions have been done. Rearranging the invariant,

$$C \geq 2n + P - 2|U|$$

. This means that the total number of comparisons for both phases satisfies

$$C + |U| - P - 1 \geq 2n + P - 2|U| + |U| - P - 1 = 2n - |U| - 1.$$

Since $L = \lfloor n/2 \rfloor$, $U \leq \lceil n/2 \rceil$. Therefore,

$$C + |U| - P - 1 \geq \lceil \frac{3n}{2} \rceil - 2,$$

which completes the proof. ∎

3.4. References.

Expected time bounds are discussed in [FlR75]. Reiser [Rei78] discusses the selection problem when the elements have weights. Further lower bounds for the selection problem are proven in [Hya76].

Minimum Spanning Trees

4.1. A Schema for Minimum Spanning Tree Algorithms.

Let $G = (V, E)$ be a connected undirected graph with $|V| = n$ nodes and $|E| = e$ edges, and let $c(v, w) \geq 0$ be a cost value attached to $(v, w) \in E$.

Definition:

A *minimum spanning tree* for (G, c) is any tree $T = (V, E')$ with the same node set as G, and with edge set $E' \subseteq E$ such that $\sum_{(v,w)\in E'} c(v, w)$ is minimum.

First basic observation:

Let $V' \subseteq V$, and let (v, w) be an edge in E, such that $v \in V'$ and $w \in V - V'$ and such that $c(v, w)$ is minimum with respect to all edges having the first property. Then some minimum spanning tree for (G, c) contains (v, w).

Second basic observation:

Let (v, w) be an edge on some cycle C in G such that $c(v, w)$ is maximum for all the edges on C. Then some minimum spanning tree for (G, c) does not contain (v, w).

The proof of these two properties is left as an easy exercise. Though both of them can be used to construct efficient algorithms for finding minimum spanning trees, we shall, in the sequel, only employ the first. From it, we can immediately derive the following very general frame for minimum spanning tree algorithms:

1. *Initialize* a forest F of n trees, each of them a singleton node from V.

Repeat as long as F has more than one tree:

2. *select* a tree T from F;
3. *find* an edge (v, w) with minimal $c(v, w)$ such that v is in T and w is not; let T' be w's tree;
4. *update* F by combining T, T', and (v, w) into a single tree.

There are a number of possibilities to implement the forest F, to choose criteria for the selection of a tree T in step 2, and to facilitate the search operations in step 3. We are going to discuss two solutions. The first one is very simple and easy to program, whereas the second has an asymptotic complexity which is the best known so far for the general case (better upper bounds are known if we restrict the class of graphs being considered, e.g., to *planar* graphs).

4.2. Kruskal's Algorithm.

In Kruskal's algorithm [Kru56], the edges are first sorted according to increasing cost, and then searched in this order. Whenever an edge is encountered which connects two different trees, these two trees are combined into one. For this last operation, a simple *weighted union* algorithm is used. We assume now for convenience that the nodes of the graph G are just the integers between 1 and n.

```
algorithm MST1;
begin
  generate a list L containing the edges of G sorted according to nondecreasing cost value;
  initialize an in-forest F, with the i-th tree T_i consisting of node i, for 1 ≤ i ≤ n
  co an in-tree is a rooted tree with all edges pointing towards the root oc;
  initialize MST to the empty tree;
  for i := 1 to length(L) do
  begin
    (v,w):= i-th element of L;
    x := j where j such that v is a node in the tree T_j of F;
    y := j where j such that w is a node in the tree T_j of F;
    if x ≠ y then
    begin
      MST := MST ∪ {(v,w)};
      if |T_x| ≤ |T_y| then UNION(T_y,T_x) co making the root of T_x a new son of the root of T_y oc
      else UNION(T_x,T_y)
    end
  end
end MST1.
```

Time complexity analysis for MST1:

1. The initial sorting of the edges takes time $\Theta(e \log e) = \Theta(e \log n)$.
2. The UNION operations can obviously be done each in time $O(1)$.

In order to analyse the complexity of the FIND operations which determine the tree some node is currently in, we notice that at all time steps and for all trees T in the forest F, $height(T) \leq \log|T|$. This is certainly true right after the initialization of F. When two trees T and T' are combined (call the result for the moment U and assume without loss of generality that $|T| \leq |T'|$), we have

$$\begin{aligned} height(T'') &\leq \max\{height(T'), height(T) + 1\} \leq \max\{\log|T'|, 1 + \log|T|\} \\ &\leq \log(|T| + |T'|) = \log|T''|. \end{aligned}$$

3. It follows hence that the FIND operations can be done in time $O(\log n)$ each.

Together, we obtain $\Theta(e \log n)$ as an upper and also as a worst-case lower bound for the running time of MST1.

4.3. A Better Minimum Spanning Tree Algorithm.

We are able to improve on the complexity of the previous algorithm by applying some care in the selection of the tree in step 2 of the general algorithm. The selection rule which we are going to discuss is called *uniform selection* (used by Sollin, see [BGH56]), and is specified as follows:

Initially, all of the n trees (each consisting of a singleton node) are placed on a queue.
Also, the *stage* of each of these trees is defined to be 0.

In step 2 of the general algorithm, the tree T at the front end of the queue is selected (it will always have a minimal *stage* among all the trees in the queue). When it is combined with some tree T' in the

queue, both T and T' are removed from the queue, and the combined tree is inserted at the end of the queue. Its *stage* is defined to be one more than the *stage* of T (which is the minimum of the *stages* of T and T').

Basic observations about uniform selection:

a) The *stage* of any tree in the queue is $\leq \log n$.

b) At any time step of the algorithm, all trees in the queue with minimal *stage* are pairwise disjoint.

The first fact comes from the observation that, in order to create an additional tree of *stage* $i+1$, it takes two trees of *stage* i. For the second property, one can easily prove by induction that at every time step the trees in the queue form a partition of the node set of the graph G.

[An alternative tree selection rule is called *smallest tree selection*. Under it, the trees are also organized in a queue, but at every step a tree with a minimal number of nodes is chosen. In an efficient implementation, one has to take care that such a minimal size tree can be found quickly, and that the combined tree can easily be reinserted. It would also be possible to use this selection rule in the following algorithm, but a somewhat bigger programming effort would result.]

For step 3 of the general algorithm, we implement the set of edges incident to some tree T (which in turn is represented by its set of nodes) as a collection of ordered subsets of size $\leq k$, where k will be specified later. Specifically, if there are m such edges, we divide them into $\lceil m/k \rceil$ subsets, $\lfloor m/k \rfloor$ of them of size k, and sort all these subsets in order of nondecreasing cost. We also attach to the head of each subset a pointer to the tree to which it belongs, and a list pointer pointing to the smallest cost edge which has not yet been searched. Furthermore we assume that from every edge (v, w) there are pointers to the head of its subset and to its twin representation (w, v) (occurring in a list attached to w's tree).

```
algorithm MST2;
begin
  procedure combine_upto_stage(s);
  begin
    while stage of the first tree in the tree queue is ≤ s and the tree queue has length ≥ 2 do
    begin
      let T be the first tree in the queue;
      search each of T's edge lists in the direction of increasing cost, starting in each list at the smallest
      edge not yet searched, and deleting all edges until one is found that connects T to a different tree,
      say T';
      add this edge to MST;
      combine T's and T''s edge lists by updating the tree pointers in T's lists so that they point to T';
      remove T and T' form the queue;
      insert the combined tree at the end of the queue, and
      set its stage to the stage of T plus one
    end
  end combine;
  MST := ∅;
  initialize the tree queue: each tree consists of a singleton node and is assigned stage 0;
  for every such tree, represent the set of incident edges by a collection of lists of size 1;
* combine_upto_stage(log log log n);
* for every tree in the tree queue, reinitialize the collection of edge lists into ordered lists of size log log n;
  combine_upto_stage(log log n);
  for every tree in the tree queue, reinitialize the collection of edge lists into ordered lists of size log n;
  combine_upto_stage(log n)
  co by the first basic observation, this implies that only one tree is left in the tree queue oc
end MST2.
```

Time complexity analysis for MST2:

Define *stage* s of the algorithm to be the time span during which the trees taken from the front end of the tree queue have *stage* s, and note that

1. if the edge lists are (re)initialized right before *stage* s, then during *stage* s there are at most $n/2^s$ trees in the tree queue, and hence at most $2e/k + n/2^s$ edge lists for all trees together;
2. the time for executing stage $s' \geq s$ (without intermediate reinitialization) is

$$O(\frac{e}{k} + \frac{n}{2^s} + D_{s'}),$$

where $D_{s'}$ is the number of edges deleted during stage s'. Because of the second basic property stated above, the trees which are in the tree queue at the beginning of stage s' are all pairwise disjoint, and all edge lists attached to them are searched at most once during stage s', because of the uniform selection rule. Also, due to the chosen implementation, the tree(s) to which an edge is incident can be found in time $O(1)$, and the time to combine the tree T (selected from the front end of the queue) with some other tree T' is proportional to the number of edge lists attached to T;

3. the time for (re)initializing the edge lists to ordered lists of size k is bounded by $O(e \log k)$. Hence,
4. the total time to (re)initialize and then execute from stage s_1 to stage s_2 is bounded by

$$O\left(e \log k + \left(\frac{e}{k} + \frac{n}{2^{s_1}}\right)(s_2 - s_1 + 1) + e\right)$$

where the first term accounts for the initialization, the second for the UNION operations, and the second and third together for the FIND, MIN and edge deletion operations.

5. Of the three phases of algorithm MST2, the third phase (between stages $\log \log n$ and $\log n$ produces the dominating contribution. Substitution of its parameters in the above estimate yields

$$O(e \log \log n + (e/\log n + n/2^{\log \log n})(\log n - \log \log n) + e).$$

Theorem:

Algorithm MST2 finds a minimum spanning tree for an arbitrary connected undirected graph with weighted edges within time bounded by $O(e \log \log n)$.

The steps marked (*) in Algorithm MST2 could actually be omitted without affecting the growth rate of its time complexity (producing, however, a bigger constant factor in the above estimate). Algorithm MST2 was originally discussed in [Ch176]. This reference also presents a number of alternative implementations and more efficient algorithms for special subcases of the minimum spanning tree problem. The first solution of time complexity $O(e \log \log n)$ for the general problem was given in [AYa75].

Path Compression

5.1. The UNION-FIND Problem.

For most of the groups of operations on sets which we have seen so far, time $O(\log n)$ per operation is required in the worst case. If we consider only the two operations UNION and FIND we can do better using a different tree structure (namely, in-trees).

The UNION-FIND algorithm has numerous applications, e.g. for the following problems: determination and verification of minimum spanning trees, depth determination, closure of equivalence relations, dominators in flow graphs, and many others [Tar79].

In general, we are given a sequence of n intermixed UNION and FIND operations as defined in Section 2. We may assume here without loss of generality that the universal set is $\{1, \ldots, n\}$, and the set names are also elements of $\{1, \ldots, n\}$. Also note that the argument sets of the UNION operations are always disjoint.

The basic idea is to construct an *in-tree* (i.e., the edges point towards the root of the tree) for each set made up of nodes for each element in the set. Each node is a pair consisting of an integer representing the element, and a pointer to another pair (its father in the tree). For the root of the tree, this pointer is assumed to be nil. Without loss of generality, the root of the tree is identified with the name of the set. The set operations are implemented as follows:

- FIND(i): from i's node, follow the chain of pointers until the next pointer is a nil-pointer; its node is the name of i's set.
- UNION($name_1, name_2$): if the two nodes are different, make the second node a new son of the first node (i.e., the nil-pointer of $name_2$ is updated to point to $name_1$).

For this implementation the following sequence obviously is a bad case:

UNION(2, 1);UNION(3, 2);...;UNION($n, n-1$); n-times FIND(1);

From it we can immediately derive the following

Theorem:

The above implementation of UNION-FIND has time complexity $\Theta(n^2)$.

5.2. The Weighting Heuristic.

There are two heuristics which can be used to improve the basic algorithm. The first is the weighting heuristic and is motivated by the desire to keep the trees balanced. We add to each root a counter which gives the number of nodes in the tree. The weighting heuristic states that when two trees are joined the root of the one with the smaller node count is made a new son of the root of the other. (In case of a draw, we still make the root of the second tree a new son of the root of the first tree.) As we have proved

already in the analysis of the minimum spanning tree algorithm MST1 (Kruskal's algorithm, Section 4.2) the *height* of a tree with n nodes constructed according to this weighting heuristic is bounded by $\log n$. Hence the FIND operations require time at most $O(\log n)$ each while each UNION can still be done in constant time. On the other hand, let $n = 2^k$ be a power of 2. Then the sequence

UNION(2, 1);UNION(4, 3); ... ;UNION($n, n-1$);
UNION(4, 2);UNION(8, 6); ... ;UNION($n, n-2$);
⋮
UNION($n, \frac{n}{2}$);
n-times FIND(1);

clearly requires time $O(n \log n)$. Hence we obtain the

Theorem:

The UNION-FIND algorithm with weighting heuristic has time complexity $\Theta(n \log n)$.

5.3. The Path Compression Heuristic.

The second heuristic to improve the basic UNION-FIND algorithm is called *path compression* or *collapsing*. It is motivated by the observation that in the bad case example for the basic algorithm n FIND(1) operations are performed each requiring time $O(n)$. After the first such FIND operation, the structure should be modified in such a way that subsequent FIND operations for the same element are sped up. This can be done by changing all pointers encountered in a FIND operation on the way up to the root, to point to that root. We can use a stack to store all intermediate nodes on the FIND path, or we can even use these pointer cells themselves (and some constant amount of additional memory) to keep track of the FIND path (details to be worked out as a homework assignment).

It is also left as an exercise to construct a special sequence of UNION's and FIND's (here the UNION's do not employ the weighting heuristic, rather the root of the second tree is always made a son of the root of the first tree!) which requires time $O(n \log n)$. Such a sequence is not hard to find if one has in mind the two basic ways to parse binomial trees.

5.4. Upper Bounds for UNION-FIND with Path Compression.

The derivation of good upper bounds for UNION-FIND algorithms which use the path compression heuristic requires some more effort. Assume that starting from singleton sets $n-1$ UNION's are performed, and $m \geq n$ (intermixed) FINDS's. Let T be the tree which would have been constructed by the UNION operations if there had been no intermixed FIND's. For every element v in the universal set (which without loss of generality we take to be the set $\{1, \ldots, n\}$ and whose elements we also identify with the nodes in the forest maintained by the UNION-FIND algorithm), let $h(v)$ be its *height* in T. (Note: this is a *static* quantity, solely depending on the order of the UNION's.)

Basic observations:

a) In any FIND path (in the original sequence with UNION's and FIND's intermixed) $v_1 \to v_2 \to \ldots \to v_r$, we have
$$h(v_1) < h(v_2) < \ldots < h(v_r).$$

b) Let $v \to w$ be an edge in a FIND path, w not equal to the root (i.e., w not the last node on the FIND path), and let $v \to w'$ be the edge from v after the path compression due to the FIND operation. Then $h(w') > h(w)$.

The first of these facts can be seen as follows. Assume we have a UNION(T_1, T_2) in the original sequence. Let r_1 (resp., r_2) be the root of T_1 (resp. T_2) at this moment. Then we have in the rearrangend sequence, and we may assume this by induction, a UNION(T_1', T_2') where T_1' (resp., T_2') contains the same elements as T_1 (resp., T_2), and is in fact a rearrangement of T_1 (resp., T_2) with the same root r_1 (resp., r_2). In the rearranged sequence, r_1 becomes a father of r_2, and hence $h(r_1) > h(r_2)$. Thus the same holds for the edge $r_2 \rightarrow r_1$ in a FIND path. Induction on the number of edges now completes the argument.

The second fact is an immediate consequence of the definition of path compression and the first fact.

5.4.1. Path Compression without Weighting Heuristic.

We divide the elements into disjoint groups and account for the cost of a FIND path by charging part of it to the FIND operation and part of it to the nodes on the path. Note that the cost of a FIND is roughly equal to the length of the FIND path.

Let the group C_i of elements consist of those nodes having between (bounds included) 2^{i-1} and $2^i - 1$ descendents in T, for $i = 1, \ldots, \lceil \log n \rceil$.

For every FIND path $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_r$ we charge

a) the last edge to the FIND operation;
b) an edge $v_k \rightarrow v_{k+1}$ with v_k and v_{k+1} in the same group to v_{k+1};
c) any other edge to the FIND operation.

As there are only $O(\log n)$ groups, every FIND operation is charged at most $O(\log n)$. Now assume that $v_k \rightarrow v_{k+1}$ and $v_{k'} \rightarrow v_{k+1}$ are (different) FIND path edges both charged to v_{k+1} (in different FIND's). Then v_k, $v_{k'}$, and v_{k+1} are all in the same group C_i. It follows, however, from the definition of the path compression heuristic that (in T) the subtrees rooted at v_k and $v_{k'}$ must be disjoint, and hence v_{k+1} would have at least $2 \cdot 2^{i-1}$ descendents, contradicting the definition of group C_i. Hence every node is charged only $O(1)$. Together with the matching lower bound (which is given as a homework problem) we have

Theorem:

The UNION-FIND algorithm with path compression but without weighting heuristic requires $\Theta(n \log n)$ time.

5.4.2. Path Compression and Weighted UNION.

We use a similar approach as before, with a different division into groups.

Consider the functions

$$F(n) := 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} \quad \text{(with } n \text{ 2's)} \qquad (F(0) := 0)$$

and

$$\log^* n := \begin{cases} 0 & \text{if } n \leq 1; \\ 1 + \log^*(\lceil \log n \rceil) & \text{otherwise.} \end{cases}$$

(Note: $\log^* F(n) = n$.)

Let now group C_i consist of those nodes x having $i = \log^*(h(x))$, i.e.,

$$F(i-1) < h(x) \leq F(i); \qquad \text{for } i = 1, \ldots, \log^* n,$$

and C_0 of the nodes with height zero.

The cost for every FIND path is charged in cases a) and c) as above whereas b) is replaced by

b') an edge $v_k \rightarrow v_{k+1}$ with v_k and v_{k+1} in the same group is charged to v_k. Hence every FIND operation is attributed a cost of at most $O(\log^* n)$. The charge to every v_k in C_i, $i > 0$, is at most

$F(i) - F(i-1)$ because of the second basic observation made above. As there are at most $n/2^h$ vertices in T with $h(x) = h$ (as we have shown for the *weighted* UNION operation in 4.2) we have

$$|C_i| \leq \sum_{j=F(i-1)+1}^{F(i)} \frac{n}{2^j} \leq 2\frac{n}{2^{F(i-1)+1}} = \frac{n}{2^{F(i-1)}} = \frac{n}{F(i)}.$$

Hence all vertices in C_i are charged together at most $O(n)$. Summing up all the pieces we obtain the

Theorem:

The UNION-FIND algorithm with path compression and weighted UNION requires $O(n \log^* n)$ time.

By a much more sophisticated analysis, which uses a multilevel subdivision into groups (we only used one level), one can improve the upper bound for this variant of the UNION-FIND algorithm to

$$O(m\alpha(m,n))$$

where $\alpha(m,n)$ is closely related to the "inverse" of Ackermann's function $A(m,n)$ with

$$\begin{aligned} A(m,0) &= 1; \\ A(0,n) &= 2n+1; \\ A(m+1,n+1) &= A(m,A(m+1,n)); \qquad \text{for } m,n \geq 0. \end{aligned}$$

Ackermann's function grows faster than any primitive recursive function. As a consequence, $\alpha(m,n) \leq 6$ for all "possible" values of m and n. For more details, we refer the reader to [Tar75]. In [Tar77], a matching lower bound for the UNION-FIND problem is proven for a very general class of machines. Finally, [Tar79] gives some extensions of path compression to problems which require the computation of certain functions along FIND paths. However, for the lowest common ancestor problem discussed there also compare [Har80].

Pattern Matching in Strings

6.1. Pattern Matching Problems.

We are interested in finding interesting patterns in strings of characters. There are many ways that this problem can be made more rigorous, but first some definitions. Let Σ be some finite alphabet. There are two strings of interest, the text, *text*, and the pattern, *pat*. The following notations are used:

$$text = t_1 \ldots t_n, \qquad t_i \in \Sigma$$

$$text_{i,j} = t_i \ldots t_j$$

$$pat = p_1 \ldots p_m, \qquad p_i \in \Sigma$$

$$pat^{(i)} = p_1^{(i)} \ldots p_{m_i}^{(i)}, \qquad p_j^{(i)} \in \Sigma$$

We are interested in the following problems:

1. Find the first (all) occurrence(s) of *pat* as a substring in *text*.
2. Find the first (all) occurrence(s) of any of the $pat^{(i)}$ as a substring in *text*.
3. Find the longest common substring(s) of *text* and *pat*.
4. For some (each) position, i, in *text*, find the position(s) $k \neq i$, such that the common prefix of $text_{i,n}$ and $text_{k,n}$ has maximal length. This problem is called "internal matching".
5. For some (each) position, i, in text, find the position(s) k in *pat* such that the common prefix of $text_{i,n}$ and $pat_{k,m}$ has maximal length. This problem is called "external matching".
6. Problem 2, except that the $pat^{(i)}$ are given by a regular expression.

6.2. Sets of Patterns Given by Regular Expressions.

We will consider the last problem first. Let α be a regular expression over Σ, and

$$M_\alpha = (S, s_0, f, \delta)$$

be a non-deterministic finite state automaton recognizing $L_\alpha \subseteq \Sigma^*$. Suppose that M_α has the following properties:

a) M_α may contain ϵ-transitions;
b) $|\delta(s, a)| \leq 2$, $\forall a \in \Sigma \cup \{\epsilon\}$;
c) $|S| \leq 2|\alpha|$.

Here the length of the regular expression, $|\alpha|$, is computed by counting one for a symbol from Σ that appears and one for each connective including concatenation. It is easy to construct an M_α from α with these properties by structural induction.

We now compute the sequence S_i of sets of states that M_α could be in after having read $text_{1,i}$.

```
algorithm reg_pat(α,text);
begin
  construct M_α = (S, s_0, f, δ);
  for i:=0 to n do
  begin
    if i=0 then S_0 := {s_0} else S_i := ⋃_{s∈S_{i-1}} δ(s, t_i);
    mark all s ∈ S_i;
    unmark all s ∈ S − S_i;
    Q:=queue of elements in S_i;
    while Q is not empty do
    begin
      take an element, say s, from Q;
      add each unmarked s' ∈ δ(s, ε) to S_i and Q and mark it
    end
    co we have just closed S_i under ε-transitions oc;
    if f ∈ S_i and i > 0 then mark t_i with _
    co the marked elements are the the places where an instance of an element of L_α ends oc
  end
end.
```

It is obvious that exactly every prefix, $text_{1,i}$ of $text$ ending with an underscored t_i is in L_α.

The construction of M_α can be done in time $O(|\alpha|)$. Clearly the running time of the algorithm reg_pat is bounded by

$$O(n|S|) = O(n|\alpha|),$$

because in the inner loop, every $s \in S$ is put on the queue at most once and $|\delta(s, a)| \leq 2$.

Extensions:

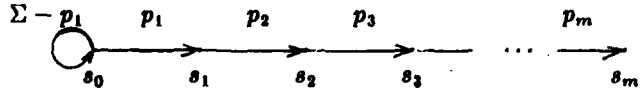
If we are looking for a pattern L_α in the middle of $text$, first replace α by $\Sigma^*\alpha$.

How to find the left end(s) of pattern(s), L_α, is left as a homework problem.

Better upper bounds can be obtained for some special forms of α. For more information see [KMP77] and [FiP74].

6.3. Automata Recognizing Substrings.

Consider for now problem 1, that of finding the first (all) occurrence(s) of pat as a substring in $text$. This is an instance of the previous problem where the regular expression α is an element of Σ^*. Consider the "skeletal machine", M'_{pat}:



General idea:

M'_{pat} reads $t_1, t_2, \ldots$ in state s_0 until the first $t_i = p_1$ is encountered. Then, M'_{pat} changes state to s_1. Suppose that after having read $text_{1,k}$, M'_{pat} is in state s_j. This implies that

$$text_{k-j+1,k} = pat_{1,j}, \text{ and}$$

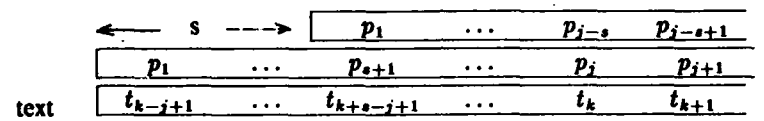
$$text_{k-j'+1,k} \neq pat_{1,j'}, \qquad \text{if } j' > j.$$

Now M'_{pat} reads t_{k+1}. If $t_{k+1} = p_{j+1}$ then M'_{pat} changes state to s_{j+1} and iterates. If $t_{k+1} \neq p_{j+1}$, then there can be no complete match at position $k - j + 1$. That is

$$text_{k-j+1,k-j+m} \neq pat.$$

We may therefore try a match at a later position by shifting the pattern $sh(j, t_{k+1})$ positions to the right. Sometimes, $sh(j, t_{k+1})$ can be greater than one, since we know that $text_{k-j+1,k} = pat_{1,j}$. In fact we can define

$$sh(j, t_{k+1}) := \min\{s > 0; text_{k+s-j+1,k+1} = pat_{1,j-s+1}\}.$$



This is always defined since if $s = j + 1$, $text_{k+2,k+1}$ and $pat_{1,0}$ are empty. For M'_{pat}, this corresponds to reading t_{k+1} and going from state s_j to state s_{j-s+1} if $t_{k+1} \neq p_{j+1}$. It is possible to build the shift function into M'_{pat}, obtaining a DFA, M_{pat}, with $O(m)$ states and $O(m|\Sigma|)$ edges.

It turns out that, it takes time $O(m|\Sigma|)$ time to construct sh, hence

Theorem:

We can determine whether pat is a substring of $text$ in time $O(m|\Sigma| + n)$.

If Σ is large the time to set up the machine might be significant. This leads to the question: is there an algorithm that is independent of $|\Sigma|$?

6.4. The Knuth-Morris-Pratt Pattern Matching Algorithm.

. To become independent of $|\Sigma|$, we make sh independent of Σ and construct a new machine M_{pat}.

$$sh'(j) := \min\{sh(j, \sigma); \sigma \in \Sigma - p_{j+1}\}.$$

Note that if $|\Sigma| < 2$ the problem is silly. Therefore,

$$sh'(j) := \min\{s > 0;\ pat_{s+1,j} = pat_{1,j-s} \text{ and } p_{j+1} \neq p_{j-s+1}\}$$

If $j \leq s$ then $pat_{1,j-s}$ and $pat_{s+1,j}$ are empty and equality holds. If $s = j + 1$ then both conditions are vacuously satisfied. This means that $0 \leq sh'(j) \leq j + 1$. If we build sh' into M_{pat}, we might, after going from state s_j to state s_{j-s}, still have a mismatch $p_{j-s+1} \neq t_{k+1}$. In this case it is necessary to iterate by reducing the state s_{j-s} according to sh'. This gives the following program:

$j := 0$; $k := 0$;
while $j < m$ and $k < n$ do
  while $j \geq 0$ and$_{cond}$ $t_{k+1} \neq p_{j+1}$ do

begin
  co $\textbf{and}_{cond}$ means that the second operand will not be evaluated unless the first operand is true **oc**;
  $j := j - sh'(j)$;
  $k := k + 1$; $j := j + 1$;
  if $j = m$ **then** "match encountered"
end.

This is the basic Knuth-Morris-Pratt algorithm. It can be refined in several ways. For details see [KMP77].

We now want to see how to construct the shift function. The basic observation is that if $sh'(0), sh'(1), \ldots, sh'(j-1)$ are known, it is easy to compute $sh'(j)$. For the following program to compute the shift function, it is useful to recall that

$$sh'(j) := \min\{s > 0;\ pat_{1,j-s} = pat_{s+1,j} \text{ and } p_{j+1} \neq p_{j-s+1}\}$$

and to consider the following figure:

The following program computes $sh'(j)$:

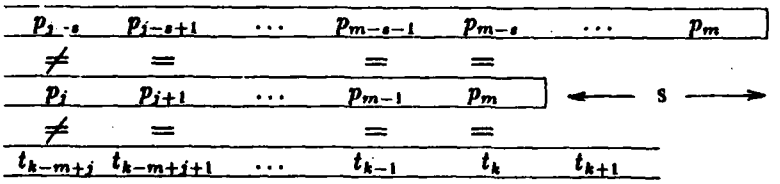
co we shift the upper copy of the pattern to the right by whatever amount is appropriate **oc**
$sh'(0) := 0$; $s := 1$; $j := 1$;
while $j < m$ **do**
begin
  while $s < j$ $\textbf{and}_{cond}$ $p_j \neq p_{j-s}$ **do**
    $s := s + sh'(j - s - 1)$;
  if $p_{j+1} = p_{j-s+1}$ **then** $sh'(j) := s + sh'(j - s)$ **else** $sh'(j) := s$;
  $j := j + 1$
end.

Whenever the inner loop of the algorithm is executed, s increases. But s is bounded from above by $m + 1$. The variable j increases once each time the outer loop is executed and is bounded from above by m. Therefore, this algorithm has time complexity $O(m)$. Similarly it holds for the main algorithm that whenever the inner loop is executed, j is decreased. But j is always at least -1 and is increased at most n times by 1. Therefore the total time for the algorithm is $O(m + n)$.

6.5. The Boyer-Moore String Matching Algorithm.

The Boyer-Moore algorithm [BoM77] uses the same basic idea as the KMP algorithm, but it compares the pattern against the text starting at the right end of the pattern and is thus often able to skip more rapidly over a part of the text that can not possibly contain a match. Two guiding rules are used to determine the shift:

a) if the next character in *text* to be compared is t, determine the rightmost position (default 0) where t occurs in *pat*;
b) choose $s > 1$ minimal such that :



It can be proved that with this shifting strategy, the Boyer-Moore algorithm has worst-case complexity (if no occurrence of the pattern is found) of $O(n)$. For a proof see [KMP77, GuO80].

On the average, the algorithm preforms much better. In this case the average is based on the assumption that all text strings of a given length are equally likely. Assume that $q := |\Sigma|$ is rather large, and let $r := \lfloor 2 \log_q m \rfloor$. Thus the probability that some $text_{k-r+1,k}$ occurs in pat is at most

$$\frac{m-r+1}{q^r},$$

because there are q^r equally likely possibilities for $text_{m-r+1}$ and at most $m-r+1$ of them occur in the pattern. If a match occurs, we need $O(m)$ steps to find all the matches in positions $k-m+1$ through $k-r$, and have shifted the pattern by $m-r$. If there is no match, we can shift the pattern by $m-r$ immediately. Hence the average time complexity is

$$O\left(\frac{n}{m-r}\left[\frac{m-r+1}{q^r}O(m) + \left(1 - \frac{m-r+1}{q^r}\right)r\right]\right)$$

$$= O\left(\frac{n \log_q m}{m}\right)$$

on a random text.

6.6. Space Efficient Linear Pattern Matching.

The pattern matching algorithms discussed so far use auxiliary space of size $\Omega(m)$ to store the values of the shift function. We now investigate a string matching algorithm *(also of linear time complexity)* that needs only a constant amount of auxiliary storage. Contrary to the KMP algorithm this algorithm must be able however, to read the input tapes (for the text and the pattern) in both directions. The algorithm is based on properties of repeating patterns in strings. We shall first give some definitions and basic lemmata, and we assume in the following that k is some fixed positive integer ≥ 4.

Definition:

a) A string z is a *period* of a string w if w is a prefix of some z^n (i.e., iff w is a prefix of zw).
b) For every $p \leq |w|$, we set
$$reach_w(p) := \max\{q; w_{1,p} \text{ is a period of } w_{1,q}\}.$$
c) A string z is *basic* if it is not of the form z'^i for any z' and $i > 1$.
d) A string z is a *prefix period* of w if z is basic and z^k is a prefix of w (i.e., $reach_w(|z|) \geq k|z|$).

Periodicity Lemma: If $|w| \geq p_1 + p_2$ and w has periods of length p_1 and $p_2 \neq p_1$ then it also has a period of length $gcd(p_1, p_2)$.

Proof: Assume without loss of generality that $p_2 > p_1$. As $w_{p_2-p_1+1,p_2} = w_{p_2+1,p_2+p_1}$ and $w_{p_1+1,|w|} = w_{1,|w|-p_1}$ because w has a period of length p_1, and as $w_{p_2+1,|w|} = w_{1,|w|-p_2}$ because w has a period of length p_2, we also have $w_{p_2-p_1+1,|w|} = w_{1,|w|-(p_2-p_1)}$, and hence w has a period of length $p_2 - p_1$. Iterating this argument as in Euclid's algorithm we obtain the lemma. ∎

Corollary: If z and z' are prefix periods of w, with $|z'| > |z|$, then $|z'| > (k-1)|z|$.

Proof: Assume to the contrary that $|z'| \leq (k-1)|z|$. Then $w_{1,k|z|}$ has periods z and z' and length $\geq |z| + |z'|$. Hence by the Periodicity Lemma, z or z' cannot be basic. But this contradicts the definition of prefix period. ∎

Decomposition Theorem: Each pattern pat can be decomposed $pat = uv$ such that v has at most one prefix period and $|u| = O(shift_v(|v|))$ where

$$shift_v(j) := \min\{s > 0; v_{s+1,j} = v_{1,j-s}\}, \text{ for } j = 1, \ldots, |v|.$$

Furthermore, such a decomposition can be found in time $O(m)$.

We postpone the proof of this Decomposition Theorem for the moment and first show how the space efficient string matching algorithm works. Basically, it checks for occurrences of the pattern suffix v in the text, and whenever it finds one naively checks for u to the left of it (here the algorithm has to back up the reading heads on the input tapes). The algorithm uses the properties of v in shifting the pattern (suffix) in case of a mismatch by an amount which is basically a fixed portion of the length of the matched prefix.

```
algorithm pmatch(text, pat);
begin
  co k is a fixed integer ≥ 4 oc;
  co for the sake of simplicity we assume that, for j = |v| + 1, v_j produces a character which does not
  occur at all in text oc;
  decompose pat = uv as stated in the Decompostion Theorem;
  i := |u|; j := 0;
  while i ≤ n − |v| and j < |v| do
  if t_{i+j+1} ≠ v_{j+1} then
  begin
    case v has a prefix period (of length r) of
      false: begin i := i + max{1, ⌈j/k⌉}; j := 0 end;
      true : if kr ≤ j ≤ reach_v(r) then
                 begin i := i + r; j := j − r end
             else
                 begin i := i + max{1, ⌈j/k⌉}; j := 0 end
  end
  else
  begin
    j := j + 1;
    if j = |v| and_cond text_{i−|u|+1,i} = u then
      "match found at position i − |u| + 1"
  end
end pmatch.
```

Proof of the correctness of the algorithm:

Again it is appropriate to imagine that the pattern (which, in this case, is the suffix v of the original pattern pat) slides on top of $text$ to the right. If $text_{i+1,i+j}$ matches $v_{1,j}$ and a mismatch occurs in the next position we can, without possibly missing a match in between, shift the pattern $shift_v(j)$ positions to the right where $shift_v(j)$ is as defined in the Decomposition Theorem:

$$shift_v(j) := \min\{s > 0;\ v_{s+1,j} = v_{1,j-s}\}.$$

In the following we show that in the algorithm *pmatch* the pattern is always shifted a distance $\leq shift_v(j)$ and hence that no occurrence of v in $text$ can be missed.

Assume first that v has no prefix period. Then the algorithm always shifts the pattern (in case of a mismatch) by $\max\{1, \lceil j/k \rceil\}$ where j is the length of the prefix of v which matches the text after the position given by the current value of i. In order for the algorithm to be correct (in this case) we must

have $j/k \leq shift_v(j)$. If we assume to the contrary that $k > k \cdot shift_v(j)$ then clearly $(v_{1,shift_v(j)})^k$ would be a prefix of $v_{1,j}$ contradicting the assumption that v has no prefix period.

Now assume that v has a prefix period of length $r \geq 1$. If $kr \leq j \leq reach_v(r)$ then certainly $shift_v(j) \geq r$ because $shift_v(j) < r$ would imply as above that v had a (second) prefix period of length $\leq shift_v(j) < r$. If $j < kr$ then $shift_v(j) < j/k$ again would imply a second, shorter prefix period. If $j > reach_v(r)$ assume again that $shift_v(j) < j/k$. Then $v_{1,shift_v(j)}$ is a period of $v_{1,j}$. Hence there is either a second prefix period for v, or $j \leq reach_v(r)$, both contradicting our assumptions.

Hence the algorithm is correct. ∎

Analysis of the running time of pmatch:

As long as we do not count the time for the decomposition of *pat* and for the checks $text_{i-|u|+1,i} = u$, the integer quantity

$$(k+1)i + j$$

increases every $O(1)$ steps (this is clear as $(k+1)(i + \lceil j/k \rceil) > (k+1)i + (k+1)j/k > (k+1)i + j$). But this quantity is bounded by $(k+1)n + m = O(n+m)$. Each time an occurrence of v is discovered in the text, the algorithm naively checks for u to the left of it using additional time $O(|u|)$ (if for this test it is necessary to first reset the input heads $|v|$ positions this time is going to be accounted for in the $O(n+m)$ time to find all occurrences of v). But v can be found at most $n/shift_v(|v|)$ times in *text*. As $|u| = O(shift_v(|v|))$ the total time for these tests is $O(n)$. Together with the bound for the decomposition of the pattern as stated in the Decomposition Theorem, we obtain the

Theorem:

The algorithm *pmatch* finds all occurrences of *pat* in *text* in time $O(n+m)$ and uses only a fixed number of (auxiliary) memory cells.

Let us finally sketch now a proof for the Decomposition Theorem.

As in the computation of the shift function for the KMP algorithm we match *pat* against itself. We actually run *pmatch* with $pat = text$ and initially $v = pat$ and u the empty string. Whenever we incur a mismatch $p_{j+1} \neq p_{i+j+1}$ we shift the upper copy of *pat* by $\max\{1, \lceil j/k \rceil\}$ until for the first time $j = ki$ holds. Up till (but not including) then no prefix period has been found so the shifting just indicated is correct as we have argued in the correctness proof for *pmatch*. When j reaches ki for the first time then $z := pat_{1,i}$ is the shortest prefix period of *pat*. We now check whether *pat* has a second prefix period (whose length then must be $> (k-1)r$, by the Corollary to the Periodicity Lemma). Note that after $i + j$ reaches $reach_{pat}(r)$, j gets decreased to zero. We continue *pmatch* with the shifting done as stated in the algorithm for the case where the pattern has a prefix period of length $r := |z|$ until for the second time j equals $(k-1)i$. If instead we reach the end of *pat* then there is no second prefix period and we can set $v := pat$ and $|u| = 0$. Otherwise the length of this second prefix period of *pat* must be greater than $reach_{pat}(r) - r$, by the Periodicity Lemma. We now cancel the initial $\lfloor reach_{pat}(r)/r \rfloor - k + 1$ copies of z from both copies of *pat*. After this, z is no longer a prefix period of the remainder *pat'*, and it follows again from the Periodicity Lemma that *pat'* cannot have a prefix period shorter than $reach_{pat}(r) - r$. Hence the *absolute* position of the pointers i and j in *pat'* does not change with respect to *pat* by the deletion of the prefix. Hence we can iterate the above process.

Note that in one iteration the length of the deleted prefix is at most the length of the next longer prefix period. Therefore, the proportion of the total length of the cancelled prefixes to the length of the last prefix period is bounded by $1 + 1/k + 1/k^2 + \ldots \leq 2$. As certainly $shift_v(|v|)$ is not less than the length of this last prefix period (v is of course the last *pat'*!) we obtain

$$|u| = O(shift_v(|v|)).$$

Finally we remark that this decomposition algorithm, for the same reasons as the main algorithm *pmatch*, runs in time $O(m)$ and uses only a fixed number of auxiliary memory. For more details and an

implementation of *pmatch* on multitape Turing machines, which runs in real-time, the reader is referred to [GaS81].

6.7. Position Trees.

Another way to solve practically all of the problems mentioned in 6.1. is to use a data structure called *position trees* , also called *prefix*, *suffix*, or *bi-tree* [AHU74, MaR80, McC76, Wei73]. Basically, for every position i in $text$ the shortest prefix p_i of $text_{i,n}$ is determined which uniquely identifies position i. This means that whenever $text$ is decomposed into up_iv then u must be $text_{1,i-1}$. The existence of such a position identifier can be guaranteed by appending to $text$ a special endmarker which does not occur elsewhere. The position identifiers p_i are then stored in a tree whose leaves are in one-to-one correspondence to the positions of $text$, and which also contains non-tree edges (also labelled with elements from the alphabet) encoding the shift function. It is pretty straightforward (though tedious) to construct position trees (respectively a compacted variant thereof) in linear time and space when reading $text$ backwards. One of the references also gives an efficient construction for the case when $text$ has to be read on-line from left to right. For more details see the references given above.

Searching Graphs and Applications

7.1. The Labelling of Trees.

For the systematic search of trees it is often useful to attach to the nodes numbers or labels which give some information about the position of a node in the tree. Of course there are many ways to specify positions in trees or graphs. We shall discuss the following standard labellings of rooted or directed trees (note that there is no essential difference: a directed tree is necessarily rooted, and a rooted tree can, in a unique way, be made directed):

- preorder numbering;
- postorder numbering;
- inorder numbering for binary trees;
- level numbering;
- descendant numbering.

We now specify these numberings in turn, by giving short pieces of programs which generate them. The variables and data structures used should be self-explicatory.

(a)
```
procedure preorder(node: v);
begin
  num := num + 1; pre[v] := num;
  for all w ∈ sons[v] do preorder(w)
end;

num := 0; preorder(root);
```

Hence, whenever w is a proper descendant of v, then $pre[w] > pre[v]$.

(b)
```
procedure postorder(node: v);
begin
  for all w ∈ sons[v] do postorder(w);
  num := num + 1; post[v] := num
end;

num := 0; postorder(root);
```

Hence, whenever w is a proper descendant of v, then $post[w] < post[v]$.

(c)
```
procedure inorder(node: v);
begin
  if v has a left son w then inorder(w);
  num := num + 1; in[v] := num;
```

```
    if v has a right son w' then inorder(w')
  end;
  num := 0; inorder(root);
```

Note that in binary trees there is a distinction between left and right sons. Let w be a descendant of the left son of v, and w' a descendant of the right son. Then we have

$$in[w] < in[v] < in[w'].$$

(d)
```
procedure level(node: v);
  begin
    for all w ∈ sons[v] do
    begin
      lev[w] := lev[v] + 1; level(w)
    end
  end;
  lev[root] := 1; level(root) co another variant sets lev[root] := 0 oc;
```

It is obvious that $lev[v]$ equals the number of nodes on the (unique) path from the root of the tree to v.

(e)
```
procedure descendants(node: v);
  begin
    for all w ∈ sons[v] do descendants(w);
    des[v] := 1 co v is considered a descendant of itself oc;
    for all w ∈ sons[v] do des[v] := des[v] + des[w]
  end;
  descendants(root);
```

Clearly, $des[v]$ counts the number of descendants of the node v.

There are many (more or less trivial) relations between these numberings. As an example we state (and leave the proof as an easy exercise) the following

Descendants Lemma: Let v and w be nodes in a (rooted) tree and assume that the *preorder* and *postorder* procedures visit the sons of every node in the same order. Then the following four conditions are equivalent:

- w is a descendant of v;
- $pre[v] \leq pre[w] < pre[v] + des[v]$;
- $post[v] - des[v] < post[w] \leq post[v]$;
- $pre[v] \leq pre[w]$ and $post[w] \leq post[v]$.

Hence we might note as a corollary that *pre* and *post* together uniquely determine the structure of the tree.

7.2. Search in Graphs.

Many search problems on graphs contain or are special variants of the following general problem:

> "Search all edges of a graph $G = (V, E)$, number the nodes in some order from 1 through $n = |V|$, and find a *spanning forest* (i.e., a set of disjoint trees with edges from E which contain all nodes in V) for the graph."

For this general problem, we give the following skeleton of an algorithm (formulated for the case of an undirected graph; the generalization to digraphs is straightforward):

while there is an unnumbered node left in the graph **do**
begin
select an unnumbered node k;
select (if there is one) an edge e between a numbered node and k, add it to the forest and perform actions_on_forest_edges(e);
for all other edges e' between numbered nodes and k **do**
actions_on_non-forest_edges(e');
number k with the next available number
end.

If we now fill in the particular rules for the selection of the node k and the edge e, we obtain a variety of graph searching methods. Some of the most important are listed below, together with their selection criteria.

BFS (*breadth-first-search*):
select the node k and the edge e such that the other endpoint of e has the lowest possible number.

DFS (*depth-first-search*):
select the node k and the edge e such that the other endpoint of e has the highest possible number.

TS (*topological search*):
select a node k with a minimal number of edges from unnumbered nodes; select e arbitrarily.

MCS (*maximum cardinality search*):
select a node k with a maximal number of edges from unnumbered nodes; select e arbitrarily.
We shall discuss applications of these search methods for graphs in the sequel.

7.3. Connectivity.

An undirected graph $G = (V, E)$ is *connected* if for any two nodes $v, w \in V$ there is a path in G from v to w. A *directed* graph is called connected if its undirected variant is connected. It is obvious how to use DFS or BFS in order to determine the connected components (i.e., the maximal connected subgraphs) of a graph.

7.3.1. Biconnectivity.

Assume that $G = (V, E)$ is a connected undirected graph. We want to know whether G can become disconnected if we just remove one of its nodes (and the edges incident on that node).

Definition:

A node $a \in V$ is called an *articulation point* of G if there are nodes $v, w \in V$ different from a such that every path from v to w passes through a. A (connected undirected) graph G is called *biconnected* if it has no articulation point.

Obviously the removal of an articulation point from a graph disconnects the graph. More general we could be interested in how many nodes it takes to disconnect a graph. The minimal number for this is called the *(node) connectivity* of the graph. Hence a connected graph with no articulation point has connectivity ≥ 2, it is *bi*connected.

Lemma:
(a) G is biconnected iff any two distinct of its edges lie on a common simple cycle.
(b) The property of lying on a common simple cycle gives rise to an equivalence relation in the edge set E. Its classes are called *biconnected components*, and their induced subgraphs *blocks*.

Proof: The proof for (b) is immediate, and the proof of (a) is given as a homework problem.

Lemma:

Let $G_j = (V_j, E_j)$ be the blocks of G.

(a) For all $i \neq j$, $|V_i \cap V_j| \leq 1$.

(b) A node a is an articulation point iff $a \in V_i \cap V_j$, for some $i \neq j$.

Proof: (a) Assume that for some $i \neq j$ there are two distinct nodes v and w in $V_i \cap V_j$. As G_i and G_j are both connected there are simple paths from v to w wholly within G_i respectively G_j and hence edge disjoint (remember that the E_j form a partition of E). But it is obvious how to construct from these two paths a simple cycle containing edges both from E_i and E_j. Contradiction.

(b) If a is an articulation point then there are (necessarily distinct) v, w such that every path from v to w passes through a. As G is connected there is at least one such simple path. Let x and y be the two nodes on this path next to a. Then the edges $\{x, a\}$ and $\{y, a\}$ must necessarily be in two different biconnected components because otherwise, by the previous Lemma, there would be a simple cycle containing these two edges, and a could not possibly be an articulation point.

On the other hand, if $a \in V_i \cap V_j$ for some $i \neq j$, then there are edges $\{x, a\} \in E_i$ and $\{a, y\} \in E_j$, and every path from x to y has to pass through a because otherwise there would be a simple cycle containing these two edges which would contradict the fact that they come from distinct biconnected components. But then a is an articulation point. ∎

We can think of a DFS being implemented recursively in the same way as the preorder labelling routine, and we say that a DFS *visits* a node v whenever the instance of the recursive search procedure for node v is active.

If we perform a depth-first-search on a connected undirected graph $G = (V, E)$ we obviously obtain a spanning forest consisting of one tree $T = (V, E')$. We also make the following basic observation about the non-tree edges in $E - E'$, from the definition of DFS:

- Every non-tree edge $\{v, w\}$ (encountered when visiting v in the DFS) is a *back-edge*, i.e., w is an ancestor of v in the tree constructed so far (and hence also in T).

This fact permits us the following notational convention: an edge $\{v, w\} \in E$ is written $v \rightarrow w$ if it is a tree edge and w is a descendant of v in the tree, and it is written $v \bar{} w$ if it is a back-edge from v to its ancestor w. We also write $v \overset{*}{} w$ for the (possibly empty) path from v to its descendant w in the tree.

Having in mind that all non-tree edges of G are back-edges and that the removal of an articulation disconnects G we immediately obtain the following characterization.

Lemma:

Let $G = (V, E)$ be a connected undirected graph, and $T = (V, E')$ a DFS-tree of G. A node $a \in V$ is an articulation point of G iff either

(a) a is the root of T and has more than one son; or

(b) there is a son s of a such that no descendant of s (including s itself) has a back-edge to a proper ancestor of a.

Proof: It is clear from the observation stated above that a is an articulation point if either one of the two conditions in the Lemma holds. Now assume that a is an articulation point in G. If a is the root of T and has only one son then G does not become disconnected if we remove a and its incident edges because all remaining nodes are reachable in the tree from this one son. Also, if a is not the root of T and every son of a has a descendant with a back-edge to a proper ancestor of a then the removal of a and its incident edges clearly does not disconnect G and a is not an articulation point. ∎

Hence, if we define the following labeling for the nodes in $v \in V$

$$low[v] := \min\{pre[x], pre[w];\ v \overset{*}{} x \bar{} w\},$$

where of course the preorder numbering is the one given by the DFS, we obtain the following

Corollary:
A node a different from the root is an articulation point iff $low[s] \geq pre[a]$ for a son s of a in T.

It is immediate to transform the definition of *low* into the following *local* form:

$$low[v] = \min\{pre[v], pre[w], low[s];\ \bar{v}\ w, s \text{ son of } v\},$$

which we can use to determine the biconnected components of G.

```
algorithm biconnect;
begin
  procedure search(node: v);
  begin
    num := num + 1; pre[v] := num; low[v] := num;
    for all edges {v, w} incident on v do
      co we assume that all these edges are given in an adjacency list of v oc
      if pre[w] = 0 then
      begin co w hasn't been visited yet in the DFS oc
        push {v, w} onto stack;
        father[w] := v
        co this information is needed to distinguish the tree edge by which w was entered oc;
        search(w);
        if low[w] ≥ pre[v] then
        begin
          declare v an articulation point if it is not the root of the DFS-tree or if it is the root and has
          at least two sons;
          pop all edges from the stack up to and including {v, w} as a new biconnected component
        end;
        low[v] := min{low[v], low[w]}
      end
      else
      if w ≠ father[v] then low[v] := min{low[v], pre[w]}
    od
  end search;
  initialize stack; num := 0; for all nodes v do pre[v] := 0;
  search(root)
end biconnect.
```

We only remark that the algorithm *biconnect* as it is formulated automatically determines the biconnected components containing edges from the root of the DFS-tree because the condition $low[w] \geq pre[root]$ $(= 1)$ is trivially satisfied whenever in $search(root)$ a $search(w)$ for a son w of the root has been performed.

Theorem:

The algorithm *biconnect* determines the biconnected components and articulation points of a connected undirected graph $G = (V, E)$ in time $O(|E|)$.

Proof: It only remains to verify the given time bound. But this is clear as the DFS visits every edge at most twice (once in every direction). ∎

Remark: A biconnected component consisting of a single edge is sometimes called a *bridge*.

7.3.2. Strongly Connected Components.

Let now $G = (V, E)$ be a *directed* graph (without multiple edges). Call two nodes $v, w \in V$ *equivalent* if there is a (directed) path in G from v to w and one from w to v (it should be clear that the relation among the nodes of G such defined is in fact an equivalence relation).

Definition:

The subgraphs induced by the equivalence classes defined by the above relation are called the *strongly connected components* (*SCC*'s) of G.

Note that SCC's are subgraphs induced by subsets of the node set of the (digraph) G (and hence we shall say that we have determined an SCC if we have determined its node set) whereas blocks in undirected graphs are given by subsets of the edge set (and hence we determined blocks by determining biconnected components).

If we perform a DFS (together with preorder numbering) on a digraph we obtain *four* kinds of edges:
- *tree edges* which are part of the spanning forest constructed by the DFS;
- *forward edges* $v \rightarrow w$ which are non-tree edges with $pre[v] \leq pre[w]$;
- *back edges* $v \rightarrow w$ where w is an ancestor of v in the DFS-forest (and hence of course $pre[w] \leq pre[v]$);
- *cross edges* $v \rightarrow w$ which are edges with $pre[w] < pre[v]$ such that w is not an ancestor of v (but has been visited before in the DFS; v and w may even be contained in different trees of the DFS-forest).

Definition:

Given an SCC C of $G = (V, E)$ and a DFS-forest of G, we define the *root* of C to be the vertex r of C with minimal preorder label.

It is immediate from the definition of the DFS routine that, if r is the root of C, the DFS visits all nodes of C between its first and last visit to r.

Lemma:

A node r is the root of an SCC iff there is no back edge from a descendant of r to a proper ancestor of r and if there is no cross edge from a descendant of r to a node w such that the root of w's SCC is a proper ancestor of r.

Proof: The proof follows immediately from the above remark and the observation that whenever one of the conditions in the lemma is not satisfied a node in r's SCC different from r has been visited before by the DFS. ∎

Again it is possible to define a labelling of G which captures the criterion of the Lemma:

Definition:

$$Lowlink[v] := \min\{pre[w];\ w \text{ can be reached from } v \text{ via zero or more tree edges possibly followed by a back edge or a cross edge to a node the root of whose SCC is an ancestor of } v\}.$$

Hence the above Lemma translates into

r is the root of an SCC iff $Lowlink[r] = pre[r]$.

And we also get a local, recursive formulation of the definition of *Lowlink*:

$$Lowlink[v] := \min\{pre[v], pre[w], Lowlink[s];\ s \text{ is a son of } v,\ v \to w \text{ is a back edge, or } v \to w \text{ is a cross edge and the root of } w\text{'s SCC is an ancestor of } v\}.$$

```
algorithm scc;
begin
  procedure search(node: v);
  begin
    num := num + 1; pre[v] := num; Lowlink[v] := num;
    push v onto stack;
    for all edges v → w do
      co we assume that all these edges are given in an adjacency list of v oc
      if pre[w] = 0 then
      begin co w hasn't been visited before oc
        search(w);
        Lowlink[v] := min{Lowlink[v], Lowlink[w]}
      end
      else
      if 0 < pre[w] < pre[v] then
        co we set pre[w] negative as soon as w's SCC has been completely determined; so here v → w
        is either a back edge or a cross edge with w still on the stack and the hence the root of w's
        SCC a proper ancestor of v oc
        Lowlink[v] := min{Lowlink[v], pre[w]}
    od;
    if pre[v] = Lowlink[v] then co v is the root of a new SCC oc
      pop all nodes on top of and including v from the stack and negate their preorder label
  end;
  initialize stack; num := 0; for all nodes v do pre[v] := 0;
  while there is a node v with pre[v] = 0 do search(v)
end scc.
```

Note that the while loop is necessary here (different from the case where the DFS was applied to a connected undirected graph in the algorithm *biconnect*) because for a digraph which is not necessarily strongly connected, DFS produces a spanning forest which may contain more than one tree.

Theorem:

The algorithm *scc* determines the SCC's of a digraph $G = (V, E)$ in time $O(|E|)$.

Proof: Again the correctness of the algorithm is obvious from the characterizations given before. And the time bound can be seen from the fact that in a digraph, DFS visits every edge exactly once. ∎

7.4. Planarity Testing.

7.4.1. Planar Graphs.

A finite undirected graph $G = (V, E)$ is called *planar* if it can be drawn in the plane (or on the surface of a three-dimensional sphere) without any edges crossing one another, i.e., the edges are

represented by connected, finite length curves, and any point common to more than one such curve must represent a node of G incident on the corresponding edges. A drawing of a graph (in the plane or on the sphere) is called *plane* if it satisfies the above planarity condition.

Planar graphs have been studied extensively in the past, both for theoretical and applied purposes. We only refer to the huge amount of work that has come from the investigation of the famous *Four Color Problem* [BeW78]. There is also a strikingly simple, classical characterization of planar graphs as given in the following

Theorem (Kuratowski, [Kur30]):

A graph is nonplanar if and only if it contains a homeomorphic preimage of K_5 or $K_{3,3}$.

Remark: K_5 is the complete graph with 5 nodes, and $K_{3,3}$ is the complete bipartite graph with 3 and 3 nodes:

K_5: $K_{3,3}$:

A homeomorphism is a *continuous* (in the topological sense) mapping. Here, a graph is considered as a manifold consisting of finite length curve segments whose intersections exactly represent the incidence relation of the graph.

For a proof of Kuratowski's Theorem we refer to [Eve79].

Though Kuratowski's criterion is very simple there seems to be no obvious way to turn it into an efficient algorithm for testing planarity of a given graph. Before we develop a, in fact linear, algorithm for this problem let's first consider a few more properties of planar graphs.

Every drawing of a graph in the plane (or on the surface of a sphere) defines (topologically) connected components which are the pieces left over when we cut along all the line segments representing the edges. For a plane drawing of a graph, these connected components are called *faces*. Note that a (finite) plane graph has exactly one unbounded face, the *outer* face, while all the other, *inner* faces are bounded.

Theorem (Euler, 1736):

Let G be a nonempty, connected plane graph with n nodes, e edges, and f faces. Then the following relation holds:

$$n + f - e = 2.$$

Proof: We prove the formula by induction on e. If $e = 0$ there is exactly one node as G is nonempty and connected, and hence there is exactly one, the outer face, and the formula holds. Assume the formula holds for all plane graphs with less than e edges, for some $e > 0$. From a plane graph with e edges, take away one edge such that either the graph remains connected or, if it becomes disconnected, one component consists of a singleton node. This can always be achieved. In the first case obviously the number of nodes remains the same whereas the number of faces decreases by one (note that the faces on the two sides of the deleted edge must have been different; otherwise, as a face is (topologically) connected, we could draw a closed line around one of the endpoints of the deleted edge without crossing any edge contradicting the fact that the graph is still connected), and by induction the formula holds. In the second case, we apply the induction hypothesis to the other connected component left after the edge

deletion. If we add to it the deleted edge and its second endpoint we do not change the number of faces while increasing the number of nodes and the number of edges each by one. Hence the formula also holds for plane graphs with e edges, completing the induction. ∎

As a consequence of Euler's formula, every plane drawing of a planar graph has the same number of faces.

Corollary:

Every planar graph without self-loops and parallel edges, and with $n \geq 3$ satisfies

$$e \leq 3n - 6.$$

Proof: Each side of an edge in a plane drawing of a graph touches only one face. As there are no self-loops and parallel edges the boundary of each face is formed by (one side of) at least three edges, and hence we obtain $f \leq 2e/3$. If the graph is not connected we may add edges without destroying its planarity such that it becomes connected. Hence we can use Euler's formula and obtain $e = n + f - 2 \leq n + 2e/3 - 2$ from which we get $e \leq 3n - 6$. ∎

Corollary:

Every planar graph without self-loops and parallel edges has a node with degree at most 5.

Proof: If we assume the contrary we obtain $6n \leq 2e$ (every edge is counted twice, once for each endpoint) which contradicts $e \leq 3n - 6$. ∎

If we wish to test planarity of graphs we may actually restrict ourselves to biconnected graphs. As we have seen in one of the homework problems the articulation points of a graph connect its blocks in form of a tree. On the other hand it is always possible to draw a planar graph in the plane in such a way that a specified node touches the outer face (the easiest way to see this is to look at a "plane" drawing of the graph on the surface of the sphere where all faces are bounded. If we choose an interior point of a face adjacent to the specified node, and then project, by a central projection with this point as center, the surface of the sphere onto a plane tangent to the sphere on the opposite side, we obtain a plane drawing of the graph with the specified node on the perimeter of the drawing). Hence, if we have a planar drawing of a block, we can find planar drawings (if they exist at all) of its children in the tree of blocks such that for each of them the connecting articulation point lies on the outside, and we can attach these drawings to the drawing of the articulation points in the father block and still obtain a plane drawing. We therefore assume in the sequel that $G = (V, E)$ is a biconnected undirected graph without self-loops and parallel edges (which also could be added later to a plane drawing).

Let $C = (V', E')$ be a (simple) cycle in G, and let E'' be $E \cap ((V - V') \times (V - V'))$.

Definition:

(a) A connected component of $(V - V', E'')$ together with the nodes in V' linked to this connected component by an edge in $E - E'$ and together with these edges is called a *bridge* of G with respect to C. The nodes both in C and the bridge are called the *attachments* of the bridge.
(b) An edge in $E - E'$ with both endpoints on C is called a *singular bridge.*

Definition:

Two bridges B and B' (of a graph G with respect to a cycle C in G) *interlace* if
(a) B has attachments a, b, the bridge B' has attachments c, d, all four are distinct, and they appear on C in the order a, b, c, d; or if

(b) B and B' have at least three attachments in common.

Lemma:

Let $B_1, \ldots, B_r$ be a set of bridges (with respect to cycle C) such that no two of them interlace and such that $C + B_i$ (the subgraph of G given by the nodes in C or B_i) is planar, for every $i = 1, \ldots, r$. Then $C + B_1 + \ldots + B_r$ is planar.

Proof: In a plane drawing clearly every bridge must be drawn completely on the inside or the outside of (the drawing of) C. Let the nodes on C be in order $v_1, \ldots, v_p, v_1$ and assume inductively that the Lemma holds for any set of $r - 1$ (non-interlacing) bridges, $r > 1$. As the bridges $B_1, \ldots, B_r$ do not interlace we can find a bridge B_j such that, if v_l is the lowest and v_h the highest (in the above order of the nodes on C) attachment of B_j no other bridge has an attachment v_i with $l < i < h$. By induction we can obtain a planar drawing of $C + B_1 + \ldots + B_{j-1} + B_{j+1} + \ldots + B_r$ (actually with all these bridges drawn on the same side of C) and also, by assumption, a plane drawing of B_j together with the edges on C between v_l and v_h. If $l \neq h$ we can "squeeze" the latter into the one bounded face of the first drawing which has the edges of C between v_l and v_h at its boundary, and if $l = h$ we can insert the plane drawing of B_j in any (bounded) face with v_l on its boundary and thus obtain a plane representation of $C + B_1 + \ldots + B_r$. Hence the induction is complete. ∎

Theorem:

Let $B_1, \ldots, B_r$ be the bridges of G with respect to C. Then G is planar iff

(a) $C + B_i$ is planar for every i, $i = 1, \ldots, r$; and

(b) the set of bridges can be partitioned into two subsets such that no two bridges in the same subset interlace.

Proof: If we have a plane drawing of G the partition of the bridges of G into those drawn on the inside respectively outside of C clearly satisfies the condition in the Theorem. The other direction is a consequence of the previous Lemma where we have in fact shown that every set of non-interlacing bridges can be drawn on one side of G. ∎

The planarity testing algorithm presented in the next section makes use of this characterization. It recursively tests whether $C + B_i$ is planar, and tries to partition the set of bridges with respect to C into two subsets as stated in the Theorem. In order to find C and the bridges a DFS is used.

7.4.2. The Hopcroft-Tarjan Planarity Testing Algorithm.

Assume that DFS is applied to G, and that the nodes are renamed such that $v = pre[v]$ for every $v \in V$. We also assume that during the DFS $low[v]$ (as defined in 7.3.1.) has been computed. For the proper parsing of G we use still another labelling of the nodes given by

$$low2[v] := \min\{v, low[w];\ w \neq low[v],\ \overset{*}{v}\ \bar{x}\ w\}.$$

The label $low2[v]$ essentially gives the second lowest node (but at least v itself) reachable from v along a branch in the DFS tree and a back-edge. Using these auxiliary functions we assign the following weight $c(v, w)$ to every edge (v, w) of G (the edges are now considered directed according to the DFS):

$$c(v,w) := \begin{cases} 2w & \text{if } \bar{v}\ w; \\ 2low[w] & \text{if } v \rightarrow w \text{ and } low2[w] \geq v; \\ 2low[w] + 1 & \text{if } v \rightarrow w \text{ and } low2[w] < v; \end{cases}$$

and order the edges in the adjacency lists of G (as the edges are directed now each edge occurs in exactly one such list) in order of nondecreasing weight. Using bucket sort this can be done in linear time.

We now describe an algorithm *find_path* which first determines a cycle C of G through the root of the DFS tree, and which then outputs a sequence of simple paths which together form the set of bridges of G with respect to C. Because of the way in which the adjacency lists of G have been reordered these paths are generated in a order which will help us to make efficient use of the planarity criterion of the last Theorem in the previous section.

```
algorithm find_path;
begin
  co we are given a DFS tree of the biconnected graph G together with the back-edges; each edge of
  G appears (as directed edge) in exactly one adjacency list, and the adjacency lists are reordered as
  described above; the nodes are named 1,...,n in the order they were visited in the DFS oc
  mark all edges "new";
  v := 1; C := {1};
  let v → u be the first (in adjacency list) edge leaving v; mark it "old";
  while u ≠ 1 do
  begin
    co as G is biconnected and because of the reordering of the adjacency lists we are guaranteed to
    reach the root again oc
    C := C ∪ {u}; v := u;
    u := endpoint of first ("new") edge leaving v;
    mark v → u "old"
  end;
  output the (simple) cycle C;
  co note v now is the highest numbered node on C oc
  while v ≠ 1 do
    co as G is biconnected there is only one tree edge from the root oc
    if there is a "new" edge leaving v then
    begin
      u := endpoint of first "new" edge leaving v;
      P := {v, u};
      while v < u co v → u is not a back-edge oc do
      begin
        v := u;
        u := endpoint of first "new" edge leaving v
        co such an edge exists because G is biconnected oc;
        P := P ∪ {u}
      end;
      output next path P
    end
    else v := father[v]
  od
end find_path.
```

It is clear from the properties of the DFS tree that *find_path* uses time $O(|V|)$ and first outputs a simple cycle C through node 1 (the root), and then a sequence of paths P, each of which is simple (because it consists of zero or more "new" tree edges followed by a "new" back-edge leading to a proper ancestor of P's first node, again because G is biconnected and also because there are no self-loops) and has exactly two nodes in common with previously generated paths, namely its first and last node. We summarize the properties of *find_path* which we shall use in the following

Observations:

(a) Let P be $f \to v \to \ldots \to l$ be a path generated by *find_path*. If $v \neq l$ then $l = low[v]$, and in every case l is the lowest node reachable from f via a path of the form $f\ \bar{v}\ w$ where $\bar{v}\ w$ has not been included in a previous path (automatically true if $f \neq v$).

(b) Let $P_1 : f_1^* l_1, P_2 : f_2^* l_2$ be two paths such that P_1 is generated before P_2 by *find_path*, and assume that $f_1^* f_2$, i.e., that f_2 is a descendant of f_1 in the DFS tree (possibly $f_1 = f_2$). Then $l_1 \leq l_2$. This is an immediate consequence of the fact noted above that the endpoint of a path is always the lowest point reachable via "new" tree edges followed by a "new" back-edge.

(c) The following property makes use of the auxiliary function *low2*. Let $P_1 : f \to v \to \ldots \to l$, and $P_2 : f \to w \to \ldots \to l$ be two paths generated by *find_path* with the same first and last node such that again P_1 is generated before P_2, and assume that $v \neq l$ and $low2[v] < f$. Then also $w \neq l$ and $low2[w] < f$. The reason is that otherwise $f \to w$ would have to come before $f \to v$ in the adjacency list of f, because of the definition of the *low2* function and the reordering of the adjacency lists.

Lemma:

Let B be a non-singular bridge, and i its largest attachment. Then B is entered via a tree edge from i, completely explored before *find_path* backs up to i again, and all other attachments of B are back-edges.

Proof: Because every path constructed by *find_path* terminates as soon as a back-edge is encountered, certainly no node of B which is not an attachment is visited before *find_path* backs up to i and traverses an edge $i \to w$ belonging to B. As B is non-singular $i \to w$ cannot be a back-edge and hence must be a tree edge. As B is connected by definition the DFS explores all edges belonging to it before backing up to i, and hence all the other attachments of B must be back-edges. ∎

Lemma:

Assume we have a plane drawing of $C + B_1 + \ldots + B_{l-1}$, where $B_1, \ldots, B_{l-1}$ are the bridges explored by *find_path* so far, and let $i \to \ldots \to j$ be the first path of the next bridge $B = B_l$ output by *find_path*. Then B cannot be added to the inside (resp., outside) of C if there is a back-edge of one of $B_1, \ldots, B_{l-1}$ drawn inside (resp., outside) of C and ending in a node k with $j < k < i$.

Proof: Assume that there is a back-edge ending in k with $j < k < i$, and let this back-edge be part of $B_{l'}$, $l' < l$. Also, let i' be the highest numbered attachment of $B_{l'}$. Then, because of the previous Lemma, we have $i' \geq i$. If $i' > i$ the bridges B_l and $B_{l'}$ interlace according to the first part of the definition of interlacing as j, i are attachments of B_l, the nodes k, i' are attachments of $B_{l'}$, all four of them are distinct and they occur on C in the order j, k, i, i'. On the other hand, if $i = i'$, let $i' \to \ldots \to j'$ be the first path of $B_{l'}$ produced by *find_path*. Then j' lies on C, and because of observation (b) about the properties of *find_path* we know that $j' \leq j$. Now we have again two cases. If $j' < j$ the two bridges B_l and $B_{l'}$ interlace as before by the first part of the definition. If however $j' = j$ we note that $B_{l'}$ cannot be a singular bridge because it also has the attachment k, and hence part (c) of our observations about *find_path* applies and allows us to conclude that the new bridge B_l must have a third attachment k' with $j < k' < i$. Depending now on whether $k = k'$ or not, we obtain that B_l and $B_{l'}$ interlace according to the second respectively first part of the definition.

In any case we have thus shown that under the conditions of the Lemma the new bridge B_l would interlace with some other bridge drawn on the same side of C which is impossible. ∎

We are now able to describe the final algorithm for planarity testing. After having determined the cycle C, it proceeds by adding to a plane drawing of C plane drawings the bridges in the order in which they are explored by *find_path*. As at any point the bridges drawn inside and those drawn outside of C could be swapped, the algorithm tries to draw every new bridge on the inside of C after having

flipped bridges drawn earlier and standing in the way to the outside. While *find_path* is outputting paths within one and the same bridge the algorithm checks recursively whether this bridge together with the segment of C between its lowest and highest numbered attachment is planar.

The previous Lemma gave a condition for bridges which cannot be drawn on the same side of C. In general, drawing one of such bridges on one side of C fixes some other bridges to be drawn on the other side of C or again on the same side. As bridges are determined by the back-edges to their attachments we can formalize this situation into the following

Definition:

A *block* of back-edges (to attachments on C) is a maximal set of back-edges such that putting one of the back-edges (and hence the bridge it belongs to) on one side of C implies the positions of all the other back-edges (and their bridges).

The planarity testing algorithm maintains two lists S_i and S_o with the back-edges which are currently drawn on the inside respectively the outside of C. The elements within each list are ordered according to nondecreasing value of the end nodes of the back-edges (actually it is sufficient to keep the lists of these nodes which might then of course contain repetitions). Now part of the back-edges of a block may be contained in S_i and part in S_o but within each of the two lists the elements of a block appear in a nice order.

Lemma:

Let j and i be the lowest respectively highest numbered attachment of back-edges in some block. If a back-edge has attachment k with $j < k < i$ then this back-edge belongs to the block.

Proof: We are going to prove the Lemma by induction on the number of bridges explored at any moment. The claim is certainly true as long as at most one bridge (with respect to C) has been explored (assuming that it together with C was found planar by a recursive application of the algorithm) because then all elements in S_i (S_o is empty) belong to one bridge and hence one block. Now assume that bridges $B_1, \ldots, B_{l-1}$ have been explored so far for some $l > 1$, and that the Lemma holds at the moment before the first path of the next bridge B_l is being output by *find_path*. Let this path be $i' \rightarrow \ldots \rightarrow j'$. As we may decide arbitrarily to draw B_l at this moment on the inside of C (if it turns out to be planar at all itself) the previous Lemma tells us that all bridges which have attachments k with $j' < k < i'$ and are currently drawn on the inside of C, have to be flipped to the outside. What is more, all blocks with such an attachment k are forced to be drawn on the other side than B_l. Hence we can combine all the back-edges of B_l and all edges in blocks with attachments k between j' and i' as above to form a new block. If there are only attachments k with $j' < k < i'$ from back-edges on S_o their position clearly is also determined by the position of B_l, and they belong to the same block as the back-edges of B_l. Hence we have established the claim of the Lemma for the moment after the exploration of one more bridge, and thus completed the induction. ∎

As a consequence of this Lemma we note that the back-edges belonging to one block appear contiguously on the lists S_i and S_o. As we have to be able to flip whole blocks from one side of C to the other whenever there occurs a conflict with the first path of a new bridge, we add, for each part of a block on one of the two lists S_i and S_o, a pointer from the last element of the block in the list to its first element. Using these pointers it is possible to combine the sublists of two adjacent blocks in constant time and also to find the blocks which overlap with a new path.

The algorithm is now straightforward. Whenever *find_path* outputs the first path P of a new bridge we determine in S_i and S_o the blocks which interfere with the new bridge and hence have to be combined with it into a new block. We then flip all blocks in S_i with attachments between the two endpoints of P to S_o. If after this swap there are again conflicting back-edges on S_i (because they were flipped in from S_o) we stop and declare the graph non-planar. Otherwise we insert a special marker in

S_o at the position of P's starting point and check recursively whether the new bridge together with the segment of C between the two endpoints of P (let us call the cycle consisting of P and this segment C'; it plays the role of C in the recursive step) is planar, using the same lists S_i and S_o. After the algorithm has completely explored the new bridge (and determined that it is planar) we still have to make sure that all bridges of this new bridge with respect to C' which have attachments to the segment of C between the two endpoints of P can in fact be drawn on the inside of C' (which they obviously have to if the new bridge is to be drawn on the inside of C). This can be done by flipping all those blocks with back-edges in S_o after the special marker introduced before the recursive step. If after this flipping process there are still (or again) back-edges after this marker we again stop because the graph is non-planar. Otherwise the algorithm proceeds to explore the next bridge.

Theorem:

The Hopcroft-Tarjan path addition algorithm tests whether a graph is planar in time $O(|V|)$.

Proof: Clearly the initial DFS with the renaming of the nodes and the computation of the auxiliary functions takes time linear in $|V|$ if we check during the search that in fact $|E| \leq 3|V|$ (otherwise we may stop immediately!). Time linear in $|V|$ is also sufficient for the reordering of the adjacency lists (done using a bucket sort) and the *find_path* algorithm because it is essentially another DFS. The total number of elements in the lists S_i and S_o is bounded by the number of back-edges, and hence $O(|V|)$. The union of two blocks can be done in constant time, and as the number of blocks decreases by one after each such union, there can be at most $O(|V|)$ such operations. Hence we obtain a total time of $O(|V|)$. ∎

7.5. Shortest Path Problems.

Suppose we are given a (directed or undirected) graph $G = (V, E)$, a length $d(x, y) \geq 0$ for every edge $x \rightarrow y$ (if G is undirected we assume that $d(x, y) = d(y, x)$ for all edges; we also assume for notational convenience that $d(x, y) = +\infty$ if the edge $x \rightarrow y$ is not present in G), and we want to know the length $dis[v, w]$ of a shortest path between two nodes v and w, i.e.,

$$dis[v, w] = \min\{\sum_{j=1}^{r} d(x_{j-1}, x_j);\ v = x_0 \rightarrow x_1 \rightarrow \ldots \rightarrow x_{r-1} \rightarrow x_r = w \text{ is a path in } G\}.$$

(Hence, by default, $dis[v, w] = \infty$ if there is no path $\overset{*}{v}\ w$.)

Shortest path problems are usually divided into the following categories:

(a) *single pair shortest path problem*: find the distance between a given pair of nodes;
(b) *single source shortest paths problem*: find the distance between a given node and all other nodes;
(c) *all pairs shortest paths problem*: find the distance between every pair of nodes.

It turns out that no algorithm for (a) is known which would not at the same time essentially solve problem (b).

7.5.1. Dijkstra's Single Source Algorithm.

We construct a bigger and bigger set S of nodes whose distance in G from some given node a is known.

algorithm single_source_shortest_path;
co determine the distance from node a to all $v \in V$; the graph $G = (V, E)$ is given by adjacency lists; $d[v, w] \geq 0$ is the length of edge $v \rightarrow w$ ($+\infty$ if not present) **oc**

```
begin
  S := {a}; dis[a] := 0;
  for all v ∈ V − {a} do dis[v] := d[a, v];
  while |S| < |V| do
  begin
    determine v ∈ V − S with minimal dis[v];
    S := S ∪ {v};
    for all (v, w) with w ∈ V − S do
      dis[w] := min{dis[w], dis[v] + d[v, w]}
  end
end single_source_shortest_path.
```

Theorem:

The algorithm *single_source_shortest_path* correctly determines the distance from a to every $v \in V$, in time $O(|V|^2)$.

Proof: We only have to verify the correctness of the algorithm, the time bound is immediate.

Whenever $|S| < |V|$ is checked, $dis[v]$ is the distance from a to v for all $v \in S$. This is certainly true before the first execution of the loop. Suppose now that just before some later execution, $v \in V - S$ is such that $dis[v]$ is minimal, assume that

$$a \to s_1 \to \ldots \to s_r \to v_1 \to \ldots \to v_q \to v$$

is a shortest path from a to v with $s_1, \ldots, s_r \in S$ and $v_1 \notin S$ (note that $r = 0$ and $q = 0$ are possible), and also assume now by contradiction that this path is strictly shorter than $dis[v]$. But, if $q > 0$, the length of $a \to s_1 \to \ldots \to v_1$ is at most the length of the whole path and hence $< dis[v_1] < dis[v]$, contradicting the choice of v. Hence q must be zero and $dis[v]$ is correct because it was set to $dis[s_r] + d[s_r, v]$ (resp., $d[a, v]$, if $r = 0$) in some earlier traversal of the inner loop and because $dis[s_r]$ was then correct by the induction hypothesis. ∎

Remarks:

(a) Using appropriate data structures (priority queues with update operations) we can implement algorithm *single_source_shortest_path* to run in time $O(|E| \log |V|)$ or $O(k|E| + k|V|^{1+1/k})$ for any positive integer k.

(b) A similar algorithm (though with running time $O(|V|^3)$ is possible for the single source shortest path problem where we allow arbitrary edge lengths but no negative length cycles (see homework problem).

For more information on these extensions see [Joh73].

7.5.2. The All Pairs Shortest Paths Problem

Now suppose we want to compute the length of a shortest path between all pairs of vertices. Let $v_1, \ldots, v_n$ be the nodes of G. Suppose the distance between x and y, $d[x, y]$ satisfies $d[x, y] \geq 0$. Define $c_{ij}^{(k)}$ to be the length of a shortest path from v_i to v_j containing as internal nodes only $v_{k'}$ with $k' \leq k$. A good algorithm to solve this problem (especially if the graph is dense) is Floyd's algorithm:

begin
  for all (i,j) **do** $c_{ij}^{(0)} := d[i,j]$ **co** 0 **if** $i = j$ **oc**;
  for $k := 1$ **to** n **do**
    for all (i,j) **do**
      $c_{ij}^{(k)} := \min\{c_{ij}^{(k-1)}, c_{ik}^{(k-1)} + c_{kj}^{(k-1)}\}$
end.

To see that this algorithm works it is necessary to observe that every shortest path is simple. It is then an easy induction on k to show that $c_{ij}^{(k)}$ is the length of a shortest path from i to j with internal nodes only $v_{k'}$ with $k' \leq k$. If v_k does not appear on the shortest path from i to j containing as internal nodes only $v_{k'}$ with $k' \leq k$ then $c_{ij}^{(k)} = c_{ij}^{(k-1)}$. Otherwise, v_k appears on the path exactly once.

The algorithm clearly requires $O(|V|^3)$ time.

7.5.3. Min-Plus Transitive Closure.

The all pairs shortest path problem can also be solved using matrix methods. It is possible to define a matrix product with other operations in place of addition and multiplication. Specifically we will be interested in what results when addition is replaced by the minimum operation and multiplication is replaced by addition, i.e., if $C = AB$ then $c_{ij} = \min_k\{a_{ik} + b_{kj}\}$. If A gives the length of edges in a graph, *i. e.* $a_{ij} = d[v_i, v_j]$ between nodes v_i and v_j, then A^2 using the Min-Plus product gives the minimal distance along paths consisting of two edges, A^3 three edges, and so on. Setting $A^* = \min_{i \geq 0} A^i$ turns out to be well defined. Here, A^0 is the matrix with 0 on the diagonal and $+\infty$ everywhere else. It functions as an identity matrix with respect to this product. The other powers are defined by $A^{m+1} = AA^m$ for $m \geq 0$. This matrix, A^*, is called the (Min-Plus) transitive closure of A. It is the solution to the all pairs shortest paths problem, if A is the distance matrix. We shall later prove that the complexity of transitive closure is the same as that of matrix multiplication (to within a constant factor). It is therefore interesting to look for good implementations of Min-Plus matrix multiplication. It should be noted that the $o(n^3)$ algorithms of Strassen and others for Plus-Times matrix multiplication will not work because the min operation does not have a well defined inverse.

There is, however, an $O(n^{5/2})$ algorithm for this problem using a different model of computation. This is the decision tree model. In this model different code may be executed depending on the results of each comparison. This code will depend on n, the size of the matrix given by the number of rows or columns it has. The $O(n^{5/2})$ algorithm is too complicated to be discussed here, so an $O(n^{5/2}(\log n)^{1/2})$ algorithm will be discussed instead. Both algorithms are described in more detail in [Fre76].

We wish to compute the Min-Plus product AB, where $A = (a_{ij})$ and $B = (b_{ij})$. We divide A into $n \times m$ submatrices, A_i. The value of m will be determined later. Similarly we divide B into $m \times n$ submatrices, B_i. The product, AB is $\min(A_1B_1, \ldots, A_{n/m}B_{n/m})$. If the A_iB_i have somehow been computed, computing AB takes n^3/m time since there are n/m matrices in the minimum and processing each one requires $O(n^2)$ time.

To compute A_1B_1, for example, we consider each $1 \leq r < s \leq m$. For each such pair we sort the $2n$ differences $a_{ir} - a_{is}$ and $b_{sj} - b_{rj}$, for $1 \leq i \leq n$ and $1 \leq j \leq n$. For all pairs of r and s this takes $O(m^2 n \log(n))$ time. Computing an entry, c_{ij} of A_1B_1 is the same as computing the $t(i,j)$ such that $a_{it} + b_{tj}$ is minimum. Since $a_{ir} + b_{rj} \leq a_{is} + b_{sj}$ if and only if $a_{ir} - a_{is} \leq b_{sj} - b_{rj}$, once the sort is done determining $t(i,j)$ can be done in constant time in the decision tree model since it is completely determined by the result of the sort.

Therefore computing the A_iB_i requires $O(mn^2 \log(n))$ time. If $m = n^{1/2}/(\log(n))^{1/2}$ the whole algorithm takes $O(n^{5/2}(\log n)^{1/2})$ time. If n is sufficiently large it is possible to precompute small decision trees. This results in an $O(n^3(\log\log n/\log n)^{1/3})$ time algorithm.

This algorithm also works for the boolean, Or-And, matrix product.

Yao and others have proved that under the decision tree model at least $\Omega(n^2 \log n)$ time is required to compute the transitive closure. [YAR77]

7.5.4. Boolean Matrix Multiplication, Transitive Closure

We now want to prove the equivalence of matrix multiplication and transitive closure as far as computational complexity is concerned. We will consider the boolean Or-And product here. The proof will also work for the Min-Plus product discussed earlier. To aid notation we will denote the additive operation by "$\vee$" and the multiplicative one by "$\wedge$". The Or-And product is then $c_{ij} = \vee_k a_{ik} \wedge b_{kj}$ for the product $C = AB$. To get an intuitive idea of what is going on it is useful to suppose that the matrices are adjacency matrices. Let A^* be the transitive closure of A. Then intuitively, $a_{ij} = 1$ if and only if there is a path from i to j in the graph.

Let $T(n)$ be the time to compute the transitive closure of $n \times n$ matrices. Let $M(n)$ be the time to compute the boolean product of two $n \times n$ matrices.

Theorem:

If $T(3n) \leq cT(n)$ and $M(2n) \geq 4M(n)$ then $T(n) = \Theta(M(n))$.

The assumptions are reasonable since the first one is satisfied if the transitive closure can be computed in polynomial time and the second is satisfied if matrix multiplication takes time $\Omega(n^2)$.

Proof: First we reduce M, matrix multiplication, to T, transitive closure. Suppose we want to compute $C = AB$. Let

$$L = \begin{pmatrix} 0 & A & 0 \\ 0 & 0 & B \\ 0 & 0 & 0 \end{pmatrix}.$$

Intuitively, this corresponds to a tripartite graph with nodes x_i, y_i, z_i for $1 \leq i \leq n$. There is an edge from x_i to y_j if and only if $a_{ij} = 1$ and an edge from y_i to z_j if and only if $b_{ij} = 1$. There are no other edges. It is easy to see that

$$L^* = \begin{pmatrix} I & A & AB \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix}.$$

Therefore $M(n) \leq T(3n) = O(T(n))$.

Suppose now we want to compute L^*. Without loss of generality we can assume that n is a power of 2. Subdivide

$$L = \begin{pmatrix} A & B \\ C & D \end{pmatrix}.$$

Let

$$L^* = \begin{pmatrix} E & F \\ G & H \end{pmatrix}.$$

It is then easy to verify that

$$E = (A \vee BD^*C)^*,$$

$$F = EBD^*,$$

$$G = D^*CE,$$

$$H = D^* \vee GF.$$

Consider for example the first equation. Think of L as an adjacency matrix of a graph. Partition the nodes into two sets X and Y, so that A is the adjacency matrix for the subgraph induced by X. Consider a path between two nodes in X. Partition the path every time it touches a node in X. Each piece will either be an edge between two nodes in X or will go from a node in X to a node in Y, possibly go to some other nodes in Y and go back to X. In the first case the path is represented by an entry in A.

In the second case it is represented by an entry in BD^*C. The transitive closure of $(A \vee BD^*C)$ then represents all paths between elements of X. Therefore, $E = (A \vee BD^*C)^*$, which is what we wanted.

To find L^* two transitive closures, six matrix multiplications, and two componentwise $\vee$'s all on matrices of size $n/2$ are needed. Therefore

$$T(n) \leq 2T(\frac{n}{2}) + 6M(\frac{n}{2}) + c'n^2.$$

It is then possible to prove that $T(n) \leq cM(n)$ by induction. The basis is obvious if c is large enough. By the inductive hypothesis,

$$T(n) \leq 2cM(\frac{n}{2}) + 6M(\frac{n}{2}) + c'n^2 \leq \frac{1}{4}(2c + 6 + 4c')M(n),$$

since $M(n) \geq n^2$. The induction goes through if $c \geq c/2 + 3/2 + c'$ or $c \geq 3 + 2c'$. ∎

7.5.5. The Four Russians' Algorithm for Boolean Matrix Multiplication.

We are now interested in computing the boolean matrix product. The method of section 7.5.3. will work. Warshall published one of the earliest algorithms to solve this problem [War62]. It is also possible to consider the boolean matrices to be integer matrices and use a Plus-Times product algorithm. It will then be necessary to remember that any nonzero entry is really one. Another interesting method that lends itself to vector operations was published in [ADK70]. Suppose we want to compute $C = AB$. As with the Min-Plus product we divide A into $n \times m$ submatrices and B into $m \times n$ submatrices. This time we take $m := n/\lfloor \log n \rfloor$. We assume that $m|n$. Define $C_i = A_iB_i$.

Then $C = \bigvee_{1 \leq i \leq m} C_i$ and it takes $O(n^2m)$ to compute this. Each row of A_i has $\lfloor \log n \rfloor$ elements and B_i has that many rows. Each row of C_i is a boolean combination of the rows of B_i, given by a row of A_i. For notational convenience consider all matrices to be column vectors of boolean row vectors. It is possible to compute all of the possible boolean combinations of the rows of B_i in a reasonable amount of time. The following procedure does just that:

```
procedure bcomb(integer: i);
begin
  comb[0] := [0, ..., 0];
  for j := 1 to 2^⌊log n⌋ − 1 do
  begin
    p := ⌊log j⌋;
    comb[j] := comb[j − 2^p] ∨ b_{(i−1)⌊log n⌋+p+1}
  end
end;
```

This procedure requires $O(n^2)$ time. It is used in the final algorithm:

```
algorithm Four-Russians(array: a, b, c);
begin co we assume that the matrices a, b, c are organized as vectors of rows oc;
  const l = ⌊log n⌋;
  var comb: array[0..2^l − 1] of boolean_vector;
  for i := 1 to n do c[i] := [0, ..., 0];
  for i := 1 to n div l do co we assume that l|n oc
  begin
    bcomb(i);
    for j := 1 to n do
```

```
    begin
      nc := 0;
      for k := l downto 1 do
        if a[j, k] then nc := nc + nc + 1 else nc := nc + nc;
        c[j] := c[j] ∨ comb[nc]
    end
  end
end Four-Russians.
```

The procedure *bcomb* is called m times. The inner loop is executed nm times. Each iteration requires $O(\log n)$ time to compute nc and $O(n)$ to do the vector "∨". Therefore the total time is $O(n^3/\lfloor \log n \rfloor)$. If we count n-bit vector operations as taking unit time, then bcomb requires only $O(n)$ time. The vector "∨" require only constant time each. It is also possible to read the value of nc as consecutive bits. So the algorithm requires $O(n^2/\lfloor \log n \rfloor)$ vector operations.

7.6. References.

Eve and Kurki-Suonio [EvK77] have another algorithm for determining the transitive closure that finds the strongly connected components. Their algorithm has worst complexity $O(|V|^3)$, but is useful for sparse graphs. An algorithm with average time complexity $O(|V|^2 \log|V|)$ is described in [BFM76], and one with linear expected time in [Sch78].

Maximum Matchings in Graphs

8.1. Fundamentals.

In the following we assume that $G = (V, E)$ is an undirected, connected graph with no self-loops and no parallel edges. This is no severe restriction because self-loops and parallel edges cannot be part of matchings as will become clear from the definitions below, and as the problem for several connected components can be solved by treating each connected component separately.

Definition:

(a) A set $M \subseteq E$ is called a *matching* if no two edges in M have a node in common.
(b) A matching M is called *perfect* if $|V|$ is even and $|M| = |V|/2$.
(c) A matching M is called *maximum* if there is no matching $M' \subseteq E$ with $|M'| > |M|$.

Remark:
A perfect matching is also called a 1-*factor*.

Definition:

A (simple) path

$$v_1 — v_2 — \ldots — v_{2k}$$

in G is called an *augmenting path* with respect to a matching M if v_1 and v_{2k} are *free*, i.e., not incident to any edge in M, and exactly every other edge of the path is in M.

There are some classical characterizations for whether a matching is maximum or whether a perfect matching (which is clearly maximum) exists:

Marriage Theorem (Frobenius-König-Hall-Rado):

Let $E \subseteq V_1 \times V_2$ be a relation. E contains a matching of size $|V_1|$ (and hence, if $|V_1| = |V_2|$, a perfect matching) iff

$$|E(V)| \geq |V| \qquad \text{for every } V \subseteq V_1.$$

Theorem (Tutte 1947):

A graph $G = (V, E)$ has a perfect matching iff $|V|$ is even and there is no $S \subseteq V$ such that the number of odd size components (i.e., the number of connected components having an odd number of vertices) of the subgraph induced by $V - S$, is greater than $|S|$.

Theorem (Berge 1957):

A matching is maximum iff there is no augmenting path with respect to it.

We are not going to prove the first two theorems but refer the reader to [Hal48] and [Tut47]. A proof for Berge's Theorem will follow as a corollary of one of the lemmas presented in the sequel.

If $S, T \subseteq E$ we denote by $S \otimes T$ the *symmetric difference*

$$S \otimes T = (S - T) \cup (T - S).$$

Lemma:

Let M be a matching, and let P be an augmenting path with respect to M (here we look at P as the set of its edges). Then $M \otimes P$ is a matching, and $|M \otimes P| = |M| + 1$.

Proof: The second, fourth, ... etc. edge of P is an element of M. Hence, as the first and last node of P are free, it is clear that $M \otimes P$ is again a matching. And as P contains one more edge not in M than edges in M, the size increases by one. ∎

Lemma:

Let M, N be matchings, and let $|N| > |M|$. Then $N \otimes M$ contains at least $|N| - |M|$ vertex-disjoint augmenting paths with respect to M.

Proof: In the graph $G' = (V, N \otimes M)$ every node has degree at most two because N and M are matchings. Hence every connected component of G' is either

a) an isolated vertex, or
b) a (simple) cycle with edges alternatingly in $M - N$ and $N - M$ (note that such a cycle must have even length), or
c) a (simple) path with edges alternatingly in $M - N$ and $N - M$.

Now let $C_1, \ldots, C_r$ be the connected components of G'. It is clear from the definition of the symmetric difference that $N = M \otimes C_1 \otimes \ldots \otimes C_r$. Let M_0 be M, and set, for $i = 1, \ldots, r$, $M_i := M_{i-1} \otimes C_i$. But in the sequence $|M_0|, |M_1|, \ldots, |M_r|$ only connected components in category c) and among them only those which start and end with an edge not in M (and hence have odd length) increase the number of matched edges. Now such paths are in fact augmenting paths with respect to M because, as N is a matching, their first and last node must be free under the matching M. As every such path increases the size of the matching by one according to the previous Lemma, there must be at least $|N| - |M|$ such components and hence augmenting paths. ∎

Note here that Berge's Theorem is an immediate corollary to this Lemma.

Lemma:

Let M be a matching of size r, and suppose that the cardinality of a maximum matching N is $s > r$. Then there exists an augmenting path with respect to M of length

$$\leq 2\left\lfloor \frac{r}{s-r} \right\rfloor + 1.$$

Proof: By the previous Lemma $N \otimes M$ contains $s - r$ vertex-disjoint (and hence edge-disjoint) paths. One of them must therefore contain at most $\lfloor r/(s-r) \rfloor$ edges of M, and hence has a total length bounded by $2\lfloor r/(s-r) \rfloor + 1$. ∎

Lemma:

Let P be a shortest augmenting path with respect to M, and P' an augmenting path with respect to $M \otimes P$. Then

$$|P'| \geq |P| + |P \cap P'|.$$

Proof: Let $N := M \otimes P \otimes P'$. As therefore $|N| = |M| + 2$, by the previous Lemma the difference $M \otimes N$ contains at least two vertex-disjoint (and hence edge-disjoint) augmenting paths, say P_1 and P_2. Now $M \otimes N = P \otimes P'$, and hence

$$|P \otimes P'| \geq |P_1| + |P_2| \geq 2|P|$$

because P was chosen as a shortest augmenting path.

As by definition $|P \otimes P'| = |P| + |P'| - 2|P \cap P'|$, we obtain

$$|P'| \geq |P| + 2|P \cap P'| \geq |P| + |P \cap P'|,$$

which we wanted to prove. ∎

The combination of the previous lemmas now suggests the following scheme for an algorithm to find a maximum matching:

start with matching $M_0 = \emptyset$;
compute a sequence $M_0, P_0, M_1, P_1, \ldots, M_i, P_i, \ldots$ where P_i is a shortest augmenting path with respect to M_i, and $M_{i+1} = M_i \otimes P_i$.

Of course we have, by the earlier lemmas, that $|P_i| \leq |P_{i+1}|$, but also

Lemma:

For all i and j such that $|P_i| = |P_j|$, the paths P_i and P_j are vertex-disjoint.

Proof: Assume for contradiction that i and j are a closest pair of indices such that $i < j$, $|P_i| = |P_j|$, and P_i and P_j are not vertex-disjoint. As i and j are closest, all P_k with $i < k < j$ are vertex-disjoint from both P_i and P_j. Because of this, P_j also is an augmenting path with respect to $M_{i+1} = M_i \otimes P_i$ (because none of the P_k with $i < k < j$ touches any of the vertices in P_j), which means that $|P_j| \geq |P_i| + |P_i \cap P_j|$, and hence, as $|P_i| = |P_j|$, that $P_i \cap P_j = \emptyset$ and P_i and P_j are edge-disjoint. But if P_i and P_j had a vertex v in common they would also have in common the matched edge of M_{i+1} incident on v (note that every node of P_i is incident to a matched edge in M_{i+1}!) as all P_k with $i < k < j$ did not affect v. Hence P_i and P_j are in fact vertex-disjoint, contradicting our assumption. ∎

Theorem:

Let s be the cardinality of a maximum matching. Then the sequence

$$|P_0|, |P_1|, \ldots$$

contains at most $2\lfloor s^{1/2} \rfloor + 1$ distinct numbers.

Proof: Let $r = \lfloor s - \sqrt{s} \rfloor$. As by construction $|M_r| = r$, we obtain by our estimate of the length of a shortest augmenting path that

$$|P_r| \leq 2\left\lfloor \frac{\lfloor s - \sqrt{s} \rfloor}{s - \lfloor s - \sqrt{s} \rfloor} \right\rfloor + 1 \leq 2\left\lfloor \frac{s}{\lceil \sqrt{s} \rceil} \right\rfloor + 1 \leq 2\lfloor \sqrt{s} \rfloor + 1.$$

Hence, if $i \leq r$, then $|P_i|$ is one of the odd numbers in $\{1, 3, 5, \ldots, 2\lfloor\sqrt{s}\rfloor + 1\}$ which has cardinality $\sqrt{s} + 1$, and $|P_{r+1}|, \ldots, |P_{s-1}|$ contribute at most another $s - r - 1 \leq \lfloor\sqrt{s}\rfloor$ distinct numbers, yielding a total bounded by $2\lfloor\sqrt{s}\rfloor + 1$. ∎

Hence we get the following

Refined scheme for a maximum matching algorithm:

$M := \emptyset$;
while there is an augmenting path with respect to M **do**
begin
  $l :=$ length of a shortest augmenting path with respect to M;
  find a maximal (with respect to set inclusion) set $\{Q_1, \ldots, Q_r\}$ of augmenting paths with respect to M, all of length L and all mutually vertex-disjoint;
  $M := M \otimes Q_1 \otimes \ldots \otimes Q_r$
end.

Corollary:

The loop of the above algorithm is executed at most $O(|V|^{1/2})$ times.

Proof: This is an immediate consequence of the fact that the size of a maximum matching is bounded by $O(|V|)$. ∎

We also refer the reader to [HoK73].

8.2. Maximum Matchings in Bipartite Graphs.

We now implement the above maximum cardinality matching algorithm for ***bipartite*** graphs $G = (V_1, V_2, E)$ (i.e., the node set of G is partitioned into two sets V_1 and V_2, and **all** (undirected) edges are between a node in V_1 and one in V_2). Note that an augmenting graph in a bipartite graph always connects a free node in V_1 to a free node in V_2.

We first start a simultaneous BFS at all free nodes in V_1 which at every odd step proceeds along all unmatched edges which it has reached at its foremost level, and which at every even step proceeds from a node by the (unique, if present at all) matched edge incident to that node. This simultaneous BFS detects the length of a shortest augmenting path when it reaches a free node in V_2, and at the same time determines a subgraph of G which contains all shortest augmenting paths (with respect to the current matching M):

for all $v \in V_1 \cup V_2$ **do** $label[v] := 0$;
$R := \emptyset$; $l := 1$
co at stage l of the simultaneous BFS, the nodes in R (resp., L below) are those nodes in V_2 (resp., V_1) visited for the first time by the BFS, and hence at a distance (via an alternating path) of l from a free node in V_1 oc;
for all free nodes $v \in V_1$ **do**
  for all v—$w \in E$ **do**
  begin
    $label[w] := 1$; $R := R \cup \{w\}$
  end;
while $R \neq \emptyset$ **and** R contains no free node **do**
begin
  $L := \emptyset$; $l := l + 1$;

```
  for all w ∈ R, v—w ∈ M do
    if label[v] = 0 then begin L := L∪{v}; label[v] := l end
    co note that in this step we can never reach free nodes in V1 because we are using only matched
    edges from V2; hence if label[v] = 0, then v has not been visited by the search before oc;
  R := ∅;    l := l + 1;
  for all v ∈ L, v—w ∈ E − M do
    if label[w] = 0 then begin R := R∪{w}; label[w] := l end
end;
R := the set of free nodes in R;
```

If at the end of this routine the set R is empty then obviously no augmenting path exists, and the matching M is maximum. Otherwise the subgraph given by all edges v—w with $label[w] = label[v] + 1$ contains all shortest augmenting paths as follows immediately from the properties of BFS. Furthermore, as G is supposed to be connected the above segment of the algorithm takes time $O(|E|)$.

We now determine, by a repeated DFS in the subgraph determined above, a maximal set of vertex-disjoint shortest augmenting paths. Note that their length is l. The DFS uses a stack with the obvious operations on it.

```
stack := ∅;
for all free nodes v ∈ V1 do
begin
  co we try to construct a shortest augmenting path starting from v and vertex-disjoint from all such
  paths constructed before oc
  push(v);
  while stack ≠ ∅ do
  begin
    w := top(stack);
    if there is an edge w—w' with label[w'] = label[w] + 1 then
    begin
      push(w');
      if w' ∈ R then
      begin
        co a new shortest augmenting path has been found oc
        print(stack); stack := ∅
      end;
      label[w'] = 0 co this marks w' as visited by the DFS oc
    end
    else pop
  end
end;
```

Obviously this algorithm produces a maximal set of vertex-disjoint augmenting paths of length l, and the time it requires is again $O(|E|)$. Together we have achieved an implementation which requires for every phase described in the above scheme time linear in the number of edges of G. With the bound on the number of such phases we hence obtain

Theorem:

There is an algorithm for the maximum cardinality matching problem in bipartite graphs which requires time bounded by $O(|V|^{1/2}|E|)$.

8.3. Maximum Cardinality Matching in General Graphs.

We use the same general approach as pointed out in section 8.1. for the maximum cardinality matching problem in general graphs, i.e., we try to design an algorithm consisting of phases where in each phase a maximal set of vertex-disjoint shortest augmenting paths is used to increase the size of the matching.

We are here only going to outline the basic ideas of such an implementation, and refer to [MiV80] for more details.

The algorithm performs a simultaneous BFS (starting at all nodes free under the current matching) which, as in the bipartite case, alternates between unmatched and matched edges. Each time before a new level is added to the BFS, the algorithm checks for *bridges*. A bridge is an edge such that both its endpoints have been reached by the BFS, and both in steps involving the same kind (matched or unmatched) of edges. Also, if they were reached by matched edges, the bridge must be an unmatched edge, and vice versa.

For every bridge, the algorithm then checks by backtracing whether there are, from the two endpoints of the bridge, two vertex-disjoint alternating paths to free nodes. If such paths are found then in fact a shortest augmenting path has been found, and it is erased from the graph, together with all edges incident to nodes on the path in order to avoid that any of its parts is visited by the BFS later on. The backtracing is done by running two DFS's (one from every endnode of the bridge) in parallel and in a way such that the nodes furthest down in the DFS trees are at levels in the BFS trees differing by at most one.

If the two DFS's fail to discover an augmenting path they actually discover a *blossom*, that is an odd length cycle with a maximal number of matched edges in it which became closed by the bridge. Such a blossom is shrunk to a *pseudo-node* by introducing path compression pointers from all its nodes to its *root* which is the unique node in the cycle not incident to a matched edge also in the cycle. In this way, edges in the blossom need not be traversed any more in later searches for augmenting paths. Blossoms can occur one within another. A blossom is reexpanded if an augmenting path is detected which contains the pseudo-node belonging to it.

A careful implementation of these concepts guarantees that in every phase each edge of G has to be visited only a constant number of times, and also that the overhead for all operations is bounded by $O(|E|)$. For details see [MiV80]. We obtain the following result:

Theorem:

There is an $O(|V|^{1/2}|E|)$ algorithm for the maximum cardinality matching problem in general graphs.

8.4. Maximum Weight Matching Problems.

Another generalization of matching problems considers (undirected and connected) graphs $G = (V, E)$ where every edge is assigned a weight $c(e) \geq 0$. The *weight* $c(M)$ of a matching $M \subseteq E$ is then defined to be

$$c(M) := \sum_{e \in M} c(e).$$

A matching M is called a *maximum weight matching* if there is no matching M' of (G, c) which has weight $c(M') > c(M)$.

For the maximum weight matching problem there are no results known equivalent to those presented in section 8.1. for the maximum cardinality problem. Instead, the most efficient algorithms known for this problem use the *dual* problem that arises when one formulates the maximum weight matching problem as a linear optimization problem. This dual problem serves to select a sequence of augmenting paths in order to iteratively increment the weight of the matching. For more details see [Law76]. We summarize:

Theorem:

(a) There is an algorithm for the maximum weight matching problem of time complexity $O(|V|^3)$.
(b) There is an algorithm for the maximum weight matching problem of complexity $O(|V||E|\log|V|)$.

For the first of these algorithms, we refer to [Law76], for the second to [GaM81].

Chapter 9

Maximum Flow in Networks

9.1. Flows and Cuts.

A *network* is a digraph $G = (V, E)$ with no self-loops and parallel edges (anti-parallel edges are permitted), and with two distinguished nodes, the *source* s and the *sink* $t \neq s$. In addition, every edge $e \in E$ is assigned a nonnegative capacity $c(e)$. A *flow* in G is a real function f from E such that
(i) for every edge $e \in E$ its value is bounded by the capacity of the edge: $0 \leq f(e) \leq c(e)$; and
(ii) for every node v different from s and t, Kirchhoff's law is satisfied:

$$\sum_{e=(v,w)\in E} f(e) - \sum_{e'=(w,v)\in E} f(e') = 0.$$

By the *total flow* $F = F(f)$ we mean the value of the above sum at the source s, i.e.

$$F = \sum_{e=(s,w)\in E} f(e) - \sum_{e'=(w,s)\in E} f(e').$$

It should be clear that this sum is the negative of the corresponding sum at the sink t.
A flow f in G is called *maximum* if $F(f)$ is maximum for all flows in G (with capacity c).

Definition:

A *cut* of the network G is a subset S of V such that $s \in S$ and $t \notin S$.

For a cut S, let $\overline{S}$ denote the complement $V - S$ of S, and let $E_{S,\overline{S}}$ be the subset of edges in E from nodes in S to nodes in $\overline{S}$ ($E_{\overline{S},S}$ is defined analogously).

Lemma:

For every cut S and every flow f we have

$$F(f) = \sum_{e\in E_{s,\overline{s}}} f(e) - \sum_{e\in E_{\overline{s},s}} f(e).$$

Proof: Note that if we sum

$$F_v := \sum_{e=(v,w)\in E} f(e) - \sum_{e'=(w,v)\in E} f(e')$$

over all $v \in S$ we obtain F_v which equals $F(f)$ as all the other F_v are zero by condition (ii) for flows. However, if we rearrange the terms in the sum all those $f(e)$ where both endpoints of e are in S cancel because they appear exactly twice and with opposite sign, and we are left with the right hand side of the equation claimed in the Lemma. ∎

If we set the *capacity* of a cut S to be

$$C(S) := \sum_{e \in E_{S,\overline{S}}} c(e),$$

we immediately obtain that

$$F \leq C(S) \qquad \text{for every cut } S.$$

This upper bound can in fact be achieved as stated in the following

Theorem (Ford, Fulkerson 1956):

$$\max\{F(f);\ f \text{ flow in } G\} = \min\{C(S);\ S \text{ cut in } G\}$$

The proof of this Theorem will follow from the next lemma (also see [FoF56]).

Definition:

A sequence

$$s = v_0 — v_1 — v_2 — \ldots — v_{r-1} — v_r$$

of edges $v_{i-1}—v_i$ without node repetition is called an *augmenting path to* v_r with respect to some flow f if for all i, $i = 1, \ldots, r$, either

a) $e = (v_{i-1}, v_i) \in E$ and $c'(e) := c(e) - f(e) > 0$, or

b) $e = (v_i, v_{i-1}) \in E$ and $c'(e) := f(e) > 0$.

An *augmenting path* (with respect to f) is an augmenting path to the sink t.

It is clear that if there is an augmenting path P with respect to a flow f then f can be increased along this path by $\delta = \min\{c'(e);\ e \text{ edge in } P\}$ in the following way: If an edge e in P is directed from s to t its flow value is increased by δ, and if e is directed the other way, it is decreased by the same amount. This change clearly maintains Kirchhoff's law for all nodes different from the source and sink, and it increases the total flow by δ.

Lemma:

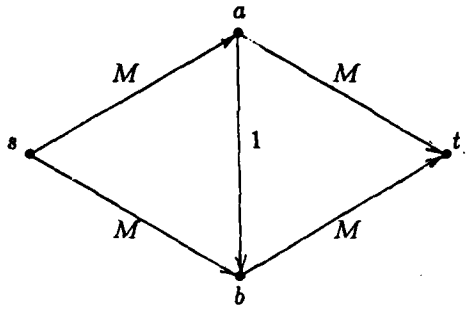
A flow is maximum iff it has no augmenting path.

Proof: It only remains to show the sufficiency of the condition in the Lemma. Hence suppose that a flow f has no augmenting path, and let S be the set of nodes v reachable from the source s by an augmenting path to v. Then $s \in S$ as the empty path is an augmenting path (to s), and $t \notin S$ as there is no augmenting path in G with respect to f. The definition of S also implies that for every edge $e \in E_{S,\overline{S}}$, the flow $f(e)$ equals the capacity $c(e)$ (because otherwise the endpoint of e would also be a member of S), and, for the same reason, that for every edge $e \in E_{\overline{S},S}$, the flow $f(e)$ is zero. Therefore we obtain, by the previous Lemma, that in fact $F(f) = C(S)$ from which we can conclude that f is maximum as, by the above Corollary, $F(f') \leq C(S)$ for every flow f'. ∎

Note that as a corollary we have also obtained a proof of the Min-Cut–Max-Flow Theorem stated earlier.

The results presented so far suggest to construct a maximum flow by starting with some flow satisfying conditions (i) and (ii), and then trying to successively increase it along augmenting paths until such a path

no longer exists in which case the resulting flow is guaranteed to be maximum. This idea also underlies the initial method by Ford and Fulkerson to construct a maximum flow. But care has to taken in selecting the augmenting paths [Zah73]. As a matter of fact, Ford and Fulkerson pointed out an example (with irrational edge capacities $c(e)$) for which the simple method (which consistently makes "bad" choices) fails to terminate and even to converge towards the actual maximum flow. But also if the edge capacities are all positive integers the edge selection is critical for the efficiency of the algorithm as demonstrated by the following example. Let M be a (very) big positive integer such that the size of its binary representation dominates the size of the whole network.



If we start in this example with a zero flow and select as augmenting paths alternatingly s—a—b—t and s—b—a—t it takes the algorithm $\Theta(M)$ augmentations to reach the maximum flow, and hence a running time exponential in the size of the input.

9.2. The Dinits Algorithm.

We now show how to overcome the difficulty indicated above by a method using so-called *pre-flows*. Call an edge v—w in the network (G, c) *augmentable from v to w* if

(a) $e = (v, w) \in E$ and $f(e) < c(e)$, or if

(b) $e = (w, v) \in E$ and $f(e) > 0$.

The following algorithm divides the network (G, c) which has some flow f defined on it, into disjoint *layers* $V_0, \ldots, V_r \subseteq V$ using a BFS:

$V_0 := \{s\}$; $\quad i := 0$;
repeat
  $T := \{v$; there is an edge w—v augmentable from w to v with $w \in V_i$ and $v \notin \bigcup_{j=0}^{i} V_i\}$;
  $i := i + 1$; $\quad V_i := T$
until $T = \emptyset$ or $t \in T$;
if $T = \emptyset$ **then** stop "flow maximum" **else** $V_i := \{t\}$;

The *layered* network $G'(G, c, f)$ now consists of all the nodes in the V_j (the final value of i is called the *length* of the network and denoted by l) and the edges in any of the E_j, $j = 1, \ldots, l$ with

$$E_j := \{v \to w;\ v \in V_{j-1},\ w \in V_j, \text{ and } v\text{—}w \in E \text{ augmentable from } v \text{ to } w\}.$$

Note that the direction of the edges in the E_j may be different from the original direction in E. In addition every edge $e = (v, w)$ in the E_j is assigned the capacity $c'(e)$ which is the sum of the values $c(e') - f(e')$ respectively $f(e')$ over all edges e' between v and w which are augmentable from v to w.

Once we have a nonzero flow f' in the layered network $G'(G, c, f)$ it is clear how to augment the flow f in (G, c): If the flow along an edge in the layered network is in the same direction as in G it is added to $f(e)$, and otherwise subtracted (a little bit of care has to be exercised if there are two antiparallel edges between some nodes v and w in (G, c, f) which are both augmentable from v to w and have been replaced by one edge in the layered network).

Definition:

A flow f' in a layered network is *maximal* if every path from s to t in the layered network contains an edge e with $f'(e) = c'(e)$.

We now propose to construct a maximum flow in a network (G, c) in phases which consist each of the following steps:

start with flow f (initially the zero flow);
construct the layered network $G'(G, c, f)$ together with the capacity c';
find a maximal flow f' in the layered network;
augment f by f'.

As the layered network $G'(G, c, f)$ contains all shortest augmenting paths with respect to f (some edges may be reversed with respect to G) we obtain a similar situation as for the maximum capacity matching problem.

Lemma:

The layered network of the k-th phase has length at least k.

Proof: The proof is by induction on k. For $k = 1$ it follows from the condition that the sink t is different from the source s. For the inductive step we note that, in the k-th phase, there can be no augmenting paths of length $< k - 1$, and also that the maximal flow constructed in the $k - 1$-st phase saturates at least one edge on each augmenting path of length $k - 1$. As all these paths appeared in the $k - 1$-st phase, no augmenting path of length $< k$ is left at the beginning of the k-th phase. But this implies that the k-th layered network has length at least k, and completes the induction. ∎

Corollary: The number of phases (as given in the above schematic algorithm) is bounded by $|V|$.

In the following, we shall be concerned with exhibiting various efficient algorithms to construct maximal flows in layered networks. Such flows are also called *preflows*.

Dinits' Method:

The first solution is due to Dinits [Din70]. It constructs a maximal flow by an iterated DFS from the source to the sink. Each successful search determines a new augmenting path, the value of the additional flow is given by the minimal capacity of an edge on the path.

```
construct G'(G, c, f) and c';
for all edges e in G' do f'[e] := 0;
while there remains an edge from the source s do
begin
  construct by a DFS a path P from s to t and delete during this DFS all dead-end edges
  co an edge is a dead-end edge if the DFS cannot proceed from its endpoint (≠ t) because it has no
  successors oc;
  Δ := min{c'[e]; e edge in P};
  for all e ∈ P do
  begin
    f'[e] := f'[e] + Δ;
    if c'[e] = Δ then delete e
  end
end.
```

Clearly the algorithm produces a maximal flow because it saturates at least one edge on every path from the source to the sink. The time requirement per phase is $O(|V||E|)$ because between any two edge deletions there are at most l steps where l is the length of the layered network, and of course $l \leq |V|$. Hence we conclude

Theorem:

Dinits' Algorithm constructs a maximum flow in an arbitrary network in time $O(|V|^2|E|)$.

9.3. The Malhotra-Pramodh Kumar-Maheshwari Algorithm.

More efficient algorithms for the maximum flow problem are also based on the above subdivision into phases and the construction of maximal preflows. But they don't use augmenting paths. The first such algorithm is due to Karzanov [Kar74]. We shall here, however, present another algorithm of the same asymptotic complexity which is easier to describe [MPM78].

Given some preflow f' in the layered network G', let us define the *potential* $P(v)$ of some node v to be the minimum of the sum of all remaining capacities $c'(e) - f'(e)$ taken over all edges e leaving v, and the same some over all edges entering into v. Clearly, $P(v)$ is the maximum amount by which the flow through v can be increased, if we look just at v. MPM algorithm can now be described as follows:

```
while P(s) and P(t) are ≠ 0 do
begin
  let v be some node with minimal P(v);
  starting from v, push additional flow of P(v) units towards t, update the flow function accordingly,
  and delete edges which become saturated;
  in the same way, starting from v, pull additional flow of P(v) units from s;
  update P;
  while there is v ≠ s,t with P(v) = 0 do
  begin
    remove v and its incident edges;
    update P
  end
end.
```

The pushing and pulling of additional flow is done in a way such that at every node that gets touched the edges are scanned in the order in which they appear on the adjacency list, and as much additional flow is added as possible, as long as the supply lasts. In this way, at most one outgoing (resp., incoming) edge receives nonzero additional flow without becoming saturated at the same time. Hence the time bound for one pass of the outer loop is

$$O(|V| + \text{the number of edges getting saturated (and hence deleted)}).$$

As in every such pass a node gets removed, we obtain a time bound of $O(|E| + |V|^2)$ per phase, and hence the

Theorem:

The MPM maximum flow algorithm has time complexity $O(|V|^3)$.

As we remarked already earlier, the Karzanov algorithm [Kar74] has the same asymptotic complexity. A somewhat more efficient algorithm for the maximum flow problem which has time complexity $O(|V|^2|E|^{1/2})$ is described in [Che77]. Other efficient solutions can be found in [GaN79], [Shi78] (both

of complexity $O(|V||E|\log^2|V|)$, and in [Sle80] (complexity $O(|V||E|\log|V|)$). These latter solutions make use of more elaborate data structures to find maximal preflows.

9.4. Extensions and Restrictions.

We'd like to shortly mention some variations of the basic maximum flow problem. For a more detailed discussion we refer to [Eve79] and [Law76].

(a) In addition to the capacity function c we may have another function b on the edges representing a lower bound for the flow f through each edge. In this case we would demand $b(e) \leq f(e) \leq c(e)$ for every edge e, and of course Kirchhoff's law as before for every node other than the source and sink. Contrary to the situation of the basic maximum flow problem where the zero flow was a legal flow, a legal flow might not exist at all in a network with arbitrary lower bounds. However, we can reduce the test for the existence of (and the construction of) a legal flow to another maximum flow problem with zero lower bounds and size proportional to the size of the original problem. Once we have established a legal flow we may use any of the methods discussed above (with the appropriate modifications in the definition of the capacity c' of the edges in the layered network) to construct a maximum flow.

(b) Instead of looking for a maximum flow, we might be interested in a *minimum* flow. But clearly, as $F_s = -F_t$, a minimum flow from s to t is a maximum flow from t to s.

(c) Often the capacities of the edges in a network are either zero or one. Such a network (and we even allow here parallel edges) is called a 0–1-*network*. Note that in a layered network coming from a 0–1-network, every edge on an augmenting path becomes saturated. Also note that a layered network which allows a big preflow, intuitively has to be wide because every edge accomodates at most one unit of flow. Using these properties, one can derive the following

Theorem:

For 0–1-networks, the Dinits algorithm has time complexity $O(|E|^{3/2})$.

(d) A special case of 0-1–networks are those with no parallel edges (also called 0–1-networks *of type* 1). For them, we obtain

Theorem:

For 0–1-networks of type 1, the Dinits algorithm has time complexity $O(|V|^{2/3})$.

(e) An even more restricted class of 0–1-networks are those where every node has either at most one outgoing or at most one entering edge. These networks are called 0–1-networks *of type* 2. It is possible for them to derive an even better upper bound:

Theorem:

For 0–1-networks of type 2, the Dinits algorithm has time complexity $O(|V|^{1/2}|E|)$.

The basic idea for the proof is again a bound on the length of the corresponding layered networks.

9.5. Applications.

We finally state some applications of the algorithms for the maximum flow problem to other problems. Again we only list the results and refer for the details to [Eve79] and [Law76].

(a) The vertex-connectivity of a directed graph is the minimal number of nodes that has to be removed from it such that there remain two vertices a and b which are no longer connected by a path from a to b. If there is an edge from every vertex to every other vertex, we set the edge-connectivity to be $|V|-1$ by default. An analogous definition is made for undirected graphs. The determination of the vertex-connectivity of (directed or undirected) graphs can be reduced to a maximum flow problem, in which basically every node of the graph is split into two nodes for the network connected by an edge with unit capacity. The algorithm then checks, for every pair (a, b) of nodes in the graph, the flow from the node corresponding to a to that corresponding to b in the network. As the network is of type 2, this can be done in time $O(|V|^{1/2}|E|)$ for every pair, and hence in a total time of $O(|V|^{5/2}|E|)$. However, a slightly better bound can be obtained if we observe that not all pairs of nodes have to be checked. As a matter of fact it suffices to check k nodes against all others if by then we have detected two nodes for which the removal of $< k$ nodes is enough to separate them. We then get the

Theorem: The vertex-connectivity of a (directed or undirected) graph can be determined in time $O(|V|^{1/2}|E|^2)$.

(b) If we are only interested if the vertex-connectivity is at least k for some given k, we can stop each flow problem once the flow reaches k units.

Theorem: It can be determined in time $O(k^3|E| + k|V||E|)$ (that is in time $O(k|V||E|)$ if $k = O(|V|^{1/2})$) whether a graph is k-vertex-connected.

(c) The edge-connectivity of (directed or undirected) graphs is defined analogously to the vertex-connectivity. However, the reduction to a maximum flow problem is here completely trivial. Also, we have to solve only $|V|-1$ network flow problems. A further possibility is to run these network flow problems in parallel, one augmenting path at a time, until one of the flows cannot be increased any more. Thus we obtain

Theorem: The edge-connectivity c of a (directed or undirected) graph can be determined in time
(i) $O(|V||E|\min\{|E|^{1/2}, |V|^{2/3}\})$, or
(ii) $O(c|V||E|)$.

(d) As a final application, we mention the maximum cardinality matching problem in bipartite graphs. The reduction to a maximum flow problem is straightforward, and we obtain, using Dinits' algorithm, the same bound as derived in section 8.2., namely $O(|V|^{1/2}|E|)$.

Problems

PROBLEM 1:
Give some examples of algorithms which you would not consider "combinatorial".

PROBLEM 2:
Prove the following formulae for the function T defined by

$$T(n) = sT(\frac{n}{m}) + an^e, \qquad \text{if } n = m^k > 1,$$
$$T(1) = d$$

(where $a > 0$, $e \geq 0$, $s > 0$, and m an integer ≥ 2):

$$T(n) = \begin{cases} \Theta(n^e), & \text{if } s < m^e \\ \Theta(n^e \log_m n), & \text{if } s = m^e \\ \Theta(n^{\log_m s}), & \text{if } s > m^e \end{cases}$$

(assume that $T(n) = 0$ for n not a power of m).
Give an explicit solution to the above recurrence (still assuming $n = m^k$).

PROBLEM 3:
In a situation analogous to the one underlying the previous problem but where n is not necessarily a power of m, one might get a recurrence like

$$T(n) = sT(\lceil \frac{n}{m} \rceil) + an^e, \qquad \text{if } n > 1$$
$$T(1) = d.$$

Usually there are two approaches to this problem:
(1) embed a problem of size n into one of size n' where $n' = m^{\lceil \log_m n \rceil}$ is the next larger power of m;
(2) embed a problem of size n into one of size n' where $n' = \lceil \frac{n}{m} \rceil m$ is the next larger multiple of m.
Compare these approaches and the bounds on the resulting complexities!
(Hint: use the facts that

$$\left\lceil \frac{\lceil n/m \rceil}{p} \right\rceil = \left\lceil \frac{n}{mp} \right\rceil \quad \text{and} \quad \left\lceil \frac{n}{m} \right\rceil \leq \frac{n}{m} + 1.)$$

PROBLEM 4:

When analyzing the Schönhage-Strassen algorithm for fast integer multiplication we arrive at a recurrence relation for its time complexity T of the form

$$T(n) = 2T(\lceil n^{1/2} \rceil) + c \log n, \qquad \text{if } n \geq 3$$
$$T(2) = T(1) = d.$$

Solve this recurrence using the techniques presented in class (assume in light of Problem 3 that n is of an appropriate form).

PROBLEM 5:

In this problem you are asked to consider the implementation details of binomial trees and queues.

(a) Describe an implementation and show how the operation of *merging* two binomial trees would be done.

(b) Show how to *delete* the smallest element in a set represented by a binomial queue using your implementation in (a). Use the algorithm of merging binomial queues mentioned in class.

(Hint: use an eldest son, next brother representation)

PROBLEM 6:

Now we are interested in the operation of deleting some *arbitrary* element that we have a pointer to. Modify your implementation of binomial queues in Problem 5 so that it allows this and the other operations to be done efficiently. Your implementation should use only two pointers per node. Describe how the delete arbitrary operation is done.

PROBLEM 7:

Prove the Decomposition Lemma stated in class.

PROBLEM 8:

Analyse both time and space complexity of an algorithm for computing 2^{2^n} on a RAM. Use both the unit and logarithmic cost criterion.

PROBLEM 9:

Describe how to modify the Paterson–Pippenger–Schönhage median algorithm presented in class so that you obtain an algorithm for arbitrary selection (i.e., an algorithm for finding the i-th largest in an n-element set for any i with $1 \leq 1 \leq n$), and determine its complexity.

PROBLEM 10:

Let $(U, \prec)$ be a totally ordered universe, and $w(s) \geq 0$ a *weight* for every $s \in U$. Describe an efficient algorithm based on the Blum-Floyd-Pratt-Rivest-Tarjan selection algorithm presented in class, which, for any given finite $S \subseteq U$ and $w \geq 0$, determines the maximal (with respect to $\subseteq$) subset M of smallest (with respect to $\prec$) elements of S such that

$$\sum_{s \in M} w(s) \leq w,$$

and analyze its complexity (in terms of number of comparisons).

PROBLEM 11:

Suppose you are given k numbers $i_0 = 0 < i_1 < \ldots < i_k < i_{k+1} = n + 1$ and you want to select simultaneously the i_1 smallest, the $i_2, \ldots, i_k$ smallest element of the n-element set S. Develop a *multiple selection algorithm* which solves this problem using at most

$$O(n \log n - \sum_{j=0}^{k} (i_{j+1} - i_j) \log(i_{j+1} - i_j))$$

comparisons.

PROBLEM 12:

Consider the problem of finding the median of n elements on a computer with limited space. Specifically, suppose that data elements are much bigger than other things, so that there is room to store $\lceil n/2 \rceil + 1$ elements and a linear amount of other stuff that can not encode data elements. Find an algorithm subject to these constraints that finds the median while doing only linearly many comparisons, for the case where $n = 2^k - 1$.

(Hint: Use the solution to the previous problem with $i_j = 2^j - 1$ for $j = 1, \ldots, k - 2$, and $i_{j+k-2} = n - 2^j + 1$, also for $j = 1, \ldots, k - 2$, and apply it to the first $\lceil n/2 \rceil$ elements read into memory. Then iteratively input as many elements as there is space left, and discard those which cannot possibly be the median.)

PROBLEM 13:

Give in PASCAL-like notation an $O(e \log \log n)$ time implementation of the minimum spanning tree algorithm MST2 discussed in class. You may assume that the vertices of the graph are represented by the integers $1, \ldots, n$. The input data consist of an (unordered) list of triples $(v, w, c(v, w))$ which contains every edge exactly once ($(v, w) = (w, v)$).

(Warning: You may not assume that arrays are initialized in a special way!)

PROBLEM 14:

Develop a linear time minimum spanning tree algorithm for *planar* graphs! (Planar graphs are those which can be drawn in the plane without edges crossing one another.) Employ a similar setting as in the first phase of algorithm MST2 and make additional use of the following facts:

a) A planar graph with n nodes has at most $3n - 3$ edges (no parallel edges are allowed; if $n \geq 3$ the number of edges is even bounded by $3n - 6$).

b) If an edge in a planar graph is shrunk to a node the graph remains planar.

PROBLEM 15:

Construct an example which shows that your approach in the previous problem does not work for arbitrary graphs with $e = O(n)$ edges.

PROBLEM 16:

Prove an $\Omega(m \log n)$ lower bound for the *unweighted* UNION–FIND algorithm which uses path compression, for a sequence of n UNION's and $m \geq n$ (intermixed) FIND's.

(Hint: Remember the two different ways to look at binomial trees!)

PROBLEM 17:

Give, in a PASCAL-like language, an implementation of the UNION-FIND operations with weighting heuristic and path compression which uses only $O(1)$ auxiliary space (this means that you cannot use a stack for intermediate storage of the FIND path; try instead to store this FIND path *in situ*). Assume that each element is represented as an element of

type *pair*=**record** *elt*: **integer**; *next*: ↑**pair end**;

and that the pointer of the *pair* for the root of a tree points to a *pair* containing the size of the tree and a **nil**-pointer.

PROBLEM 18:

Let (G, c) be a connected, undirected graph with nonnegative edge weight c. Let C be a cycle in G, and e an edge in C such that $c(e)$ is maximal among all edges in C.

a) Prove that there is a minimum spanning tree for (G, c) which does not contain e.

b) Prove that if $c(e)$ is strictly larger than $c(e')$ for all other edges e' in C, no minimum spanning tree for (G, c) contains e.

PROBLEM 19:

Let G be a connected, undirected graph with n nodes and e edges, and let T be a (rooted) spanning tree for G. Design an algorithm of time complexity $O(e(n))$ which, for every edge $\{v, w\}$ in G and not in T, determines the *lowest common ancestor* of v and w in T (the *lowest common ancestor* of v and w is the node of T where the unique paths from the root of T to v respectively w part).

(Hint: In a *post-order* search of T you hit exactly twice a node incident to a non-tree edge of G. The second time, the lowest common ancestor is on top of the nodes you have visited so far.)

PROBLEM 20:

Let (G, c) be a connected, undirected graph with nonnegative edge weight c, and let T be a spanning tree for G. Using the results of the two previous problems, describe an $O(c(n))$ algorithm which *checks* whether T is a *minimum* spanning tree for (G, c).

PROBLEM 21:

Describe how to modify the algorithm *reg_pat* presented in class such as to find for every position j in $text$

a) the minimal i such that $text_{i,j} \in L_\alpha$;

b) the maximal i such that $text_{i,j} \in L_\alpha$.

(Hint: Associate in the simulation of the pattern automaton M_α an appropriate count with every state.)

PROBLEM 22:

Suppose $a = a_1 a_2 \ldots a_r$ is a string. A substring of a is a string c of the form $c = a_{i_1} a_{i_2} \ldots a_{i_n}$, where $0 < i_1 < i_2 < \ldots < i_n < r + 1$. That is, the letters of c occur in a in the same order but not necessarily consecutively. Find an $O(|a||b|)$ algorithm that finds the longest string that is a substring of both a and b.

PROBLEM 23:

(a) Find a linear, off-line algorithm that finds the least common ancestor (LCA) of nodes of a complete binary tree. The least common ancestor of two nodes is the node that is an ancestor of both nodes

and is a descendent of all such nodes. All nodes in a complete binary tree have zero or two children. All of the leaves of a complete binary tree are at two adjacent levels. For an off-line algorithm, the tree and all queries are available before any query must be answered. The algorithm must answer q queries about an n node tree in $O(n+q)$ time. (Hint: It is possible to use radix sort to sort a nodes in $O(n+a)$ time.)

(b) Extend your algorithm in (a) to work for balanced trees. A balanced n-node tree has depth $O(\log n)$. (Hints: The algorithm should still be linear, but the constant will depend on the constant in the depth. Conceptually, extend the tree to make it complete.)

(c) [Extra credit] Consider the set of all union trees. That is trees that are created by the UNION algorithm with the weighting heuristic and without path compression. Prove that any such tree can be made into a balanced binary tree in linear time.

(d) You are to show that the LCA problem for general trees can be reduced to that for union trees. Consider a general tree T. It has a corresponding union tree, U. The nodes of U correspond to the leaves of T. Consider an internal node, v of T. Suppose that all of the sons of v correspond to sets in U. Then modify U so that there is a set that is the union of all of the sets that correspond to sons of v. Make v correspond to the node representing this set.

(i) Show the sets corresponding to the sons of v are distinct. Therefore the procedure above is well defined.

(ii) Show that U can be computed from T in linear time.

(iii) Suppose that a and b are leaves of T that correspond to A and B in U. Let C be the least common ancestor of A and B in U. Show that U and T can be preprocessed in linear time so that the least common ancestor of a and b in T can be found in constant time, given C.

PROBLEM 24:

Prove the following characterization given in class:

A connected, undirected graph is biconnected iff every pair of distinct edges lies on a common simple cycle.

PROBLEM 25:

Prove that in a connected undirected graph the articulation points connect the blocks in form of a tree.

PROBLEM 26:

Give a nonrecursive implementation (PASCAL-like) of the DFS routine (for directed and undirected, not necessarily connected graphs) using a LIFO queue.

PROBLEM 27:

Let $G = (V, E)$ be an undirected graph. A set $C \subseteq V$ is called a *clique* iff any two distinct $v, w \in C$ are connected by an edge in E. Let C be a clique, and F a DFS-forest of G. Prove that all $w \in C$ lie on one branch of a tree in F. Do they occur contiguously on this branch? Justify your answer.

PROBLEM 28:

A *Eulerian circuit* in a connected undirected graph $G = (V, E)$ is a circuit which contains each edge of the graph exactly once.

(a) Prove that G has a Eulerian circuit iff every node has even degree (i.e., an even number of edges incident upon it).

(b) Design an algorithm of complexity $O(|E|)$ which determines a Eulerian circuit of G if there is one.

PROBLEM 29:

Give an example of a planar graph (without self-loops and parallel edges) where every node has degree at least 5.

PROBLEM 30:

(a) Give the formulas for a coordinate transformation from the surface of the sphere onto the plane which transforms the plane drawing of a (finite) graph on the sphere into a plane drawing in the plane with a prespecified outer face.
(b) Give a meromorphic transformation of the (complex) plane (i.e., a transformation of the form $f(z) = \sum_{n \geq m} a_n z^n$ for some integer m converging in some region of the plane) which achieves the same task for a plane graph given in the plane.

PROBLEM 31:

The *dual graph* of a plane graph G is given by a node for every face of G and an edge connecting two nodes for every edge of G which is on the common boundary line of the two faces corresponding to the nodes.

(a) Prove that the dual of a plane graph is planar.
(b) Prove that the dual of the dual is isomorphic to the graph itself.
(c) Prove that every plane graph has either a node of degree ≤ 2 or a face with at most 5 edges on its boundary.

PROBLEM 32:

Discuss how to obtain (using the planarity testing algorithm presented in class) an adjacency list representation of a planar graph G where for each node the incident edges appear in the adjacency list in the order given by their clockwise ordering in some plane drawing of G (modulo the starting direction).

PROBLEM 33:

a) Find an $O(|V||E|)$ algorithm to solve the single source shortest path problem if edges with negative weights, but no negative weight cycles are allowed.
b) Modify your algorithm so that it solves the single source shortest path problem or finds a negative weight cycle given a graph with possibly negative edge weights.

PROBLEM 34:

Find a linear time algorithm to 6-color a planar graph. A graph is n-colored if every node is assigned one of n colors and no two nodes connected by an edge have the same color.

PROBLEM 35:

a) Prove that any planar graph can be 5-colored. (Hint: If a vertex has degree five, contract it and two of its neighbors. Use without proof that this preserves planarity.)
b) Convert your proof into an algorithm to 5-color a planar graph. Analyze its complexity in terms of $|V|$, the number of vertices in the graph.
c) Find a linear time algorithm to 5-color a planar graph. (Hint: Use without proof the fact that every planar graph has a vertex of degree four or less or it has a vertex of degree five with four neighbors all of degree at most 11.)

PROBLEM 36:

Simulate the Hopcroft-Tarjan planarity testing algorithm on the following graph. There are 7 vertices $v_1, \ldots, v_7$. There are 15 edges. Each vertex v_i is connected to v_{i+m} for $i = 1, \ldots, 7 - m$ and $m = 1, 2, 3$. Show all phases of the algorithm.

Problems for the Closed Book Final:

PROBLEM 37:

Investigating odd-even merging the following recurrence arises:

$$f_k = 2^k - 1 - f_{k-1}, \qquad f_0 = 0.$$

What is the solution?

PROBLEM 38:

Give the position tree for the string *aaaabaaabaabab*\$.

PROBLEM 39:

Prove that an undirected graph is biconnected if and only if for every triple a, b, c of nodes there is a simple path from a to c through b.

PROBLEM 40:

(a) How would you have to define the capacity of a cut in a network with (nonnegative) lower bounds such that the Min-Cut Max-Flow Theorem holds?
(b) Give a detailed proof of a "Min-Flow Max-Cut" theorem (see what function of a cut is maximum).

PROBLEM 41:

Call the sum over the degrees of the free nodes with respect to some matching m the *deficiency* $d(M)$. Give an efficient algorithm to find a matching with minimum deficiency. You may describe a reduction to some problem discussed in class. What is the complexity of the algorithm?

PROBLEM 42:

Give a program for a version of Dijkstra's shortest path algorithm which, for some given node a, in addition to all distances $dis[a, b]$, also determines the minimal number of edges on a shortest path from a to b, for every node b. Establish the correctness of your algorithm and its time complexity.

PROBLEM 43:

Let $G = (V, E)$ be a digraph with a (nonnegative) distance function d on the edges, and suppose that we want to find, for some fixed $a \in V$, the length of a second shortest path from a to b, for every $b \in V$. (Such a second shortest path need not be simple, but of course it reaches its endnode just once! It may also have the same length as a shortest path, we only require it to be different from some shortest path and having minimal length among all such paths.)
(a) What is the length $dis2[a]$ of a second shortest path from a to a?
(b) Suppose some shortest path from a to $b \in V$ contains just one edge. What is $dis2[b]$?
(c) Give an algorithm to compute $dis2[b]$ for all $b \in V$.
(d) Determine the time complexity.

References

[ADK70] ARLAZAROV, V.L., DINITS, E.A., KRONROD, M.A., FARADZEV, I.A.: On Economical Construction of the Transitive Closure of a Directed Graph.
Soviet Math. Dokl. 11 (1970), p. 1209–1210

[AHU74] AHO, A.V., HOPCROFT, J.E., ULLMAN, J.D.: The Design and Analysis of Computer Algorithms.
Addison-Wesley, Reading, Mass., 1974

[AYa75] YAO, A.C.: An $O(|E| \log\log|V|)$ Algorithm for Finding Minimum Spanning Trees.
Information Processing Letters 4 (1975), p. 21–23

[BeW78] BEINEKE, L.W., WILSON, R.J. (EDS.): Selected Topics in Graph Theory.
London-New York-San Francisco: Academic Press 1978

[BFM76] BLONIARZ, P.A., FISCHER, M.J., MEYER, A.R.: A Note on the Average Time to Compute Transitive Closures.
MIT/LCS/TM-76, Sept. 1976

[BFP73] BLUM, M., FLOYD, R.W., PRATT, V.R., RIVEST, R.L., TARJAN, R.E.: Time Bounds for Selection.
JCSS 7 (1973), p. 448–461

[BGH65] BERGE, C., GHOUILA-HOURI, A.: Programming, Games, and Transportation Networks.
John Wiley, New York, N.Y., 1965

[Bro78] BROWN, M.R.: Implementation and Analysis of Binomial Queue Algorithms.
SIAM J. on Comput. 7 (1978), p. 298–319

[Car83] CARROLL, L.: Lawn Tennis Tournaments.
St. James's Gazette (August 1, 1883), p. 5–6. Reprinted in: The Complete Works of Lewis Carroll. New York Modern Library, 1947

[Che77] CHERKASSKY, B.: Efficient Algorithms for the Maximum Flow Problem.
Akad. Nauk USSR, CEMI, Mathematical Methods for the Solution of Economical Problems 7 (1977), p. 117–125

[ChT76] CHERITON, D., TARJAN, R.E.: Finding Minimum Spanning Trees.
SIAM J. Comput. 5 (1976), p. 724–742

[Dij59] DIJKSTRA, E.W.: A Note on Two Problems in Connection with Graphs.
Numer. Math. 1 (1959), p. 269–271

[Din70] DINITS, E.A.: Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation.
Soviet Math. Dokl. 11 (1970), p. 1277–1280

[Eve79] EVEN, S.: Graph Algorithms.
Potomac, MD: Computer Science Press, Inc. 1979

[EvK77] EVE, J., KURKI-SUONIO, R.: On Computing the Transitive Closure of a Relation.
Acta Informatica 8 (1977), p. 303–314

[Flo62] FLOYD, R.W.: Algorithm 97: Shortest Path.
C.ACM 5 (1962), p. 345

[FlR75] FLOYD, R.W., RIVEST, R.L.: Expected Time Bounds for Selection.
Comm. ACM 18 (1975), p. 165–172

[FoF56] FORD, L.R., FULKERSON, D.R.: Maximal Flow Through a Network.
Canad. J. Math. 8 (1956), p. 399–404

[Fre76] FREDMAN, M.L.: New Bounds on the Complexity of the Shortest Path Problem.
SIAM J. on Comput. 5 (1976), p. 83–89

[Gal79] GALIL, Z.: On Improving the Worst Case Running Time of the Boyer-Moore String Matching Algorithm.
CACM 22 (1979), p. 505–508

[GaM81] GALIL, Z., MICALI, S.: An $O(|V||E|\log|V|)$ Algorithm for Maximum Weight Flow.
Private Communication

[GaN79] GALIL, Z, NAAMAD, A.: Network Flow and Generalized Path Compression.
Proc. 11th Ann. ACM STOC (1979), p. 13–26

[GaS81] GALIL, Z., SEIFERAS, J.: Time-space-optimal String Matching.
Proc. 13th Ann. ACM STOC (1981), p. 106–113

[GuO80] GUIBAS, L.J., ODLYZKO, A.M.: A New Proof of the Linearity of the Boyer-Moore String Searching Algorithm.
SIAM J. Comput. 9, (1980), p. 672–682

[Hal48] HALL, M.: Distinct Representatives of Subsets.
Bull. Amer. Math. Soc. 54 (1948), p. 922–926

[Har80] HAREL, D.: A Linear Time Algorithm for the Lowest Common Ancestors Problem.
Proc. 21st Ann. Symp. on FOCS (1980), p. 308–319

[HoK73] HOPCROFT, J.E., KARP, R.M.: An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs.
SIAM J. on Comput. 2 (1973), p. 225–231

[HoT74] HOPCROFT, J., TARJAN, R.E.: Efficient Planarity Testing.
J.ACM 21 (1974), p. 549–568

[Hya76] HYAFIL, L.: Bounds for Selection.
SIAM J. Comput. 5 (1976), p. 109–114

[Joh73] JOHNSON, D.B.: Algorithms for Shortest Paths.
Ph.D. Thesis, Dept. of Computer Science, Cornell University, Ithaca, N.Y., 1973. Also: Efficient Algorithms for Shortest Paths in Sparse Networks. J.ACM 24 (1977), p. 1–13

[Kar74] KARZANOV, A.V.: Determining the Maximal Flow in a Network by the Method of Preflows.
Soviet Math. Dokl. 15 (1974), p. 434–437

[KMP77] KNUTH, D.E., MORRIS, J.H., PRATT, V.R.: Fast Pattern Matching in Strings.
SIAM J. Comput. 6 (1977), p. 323–350

[Knu76] KNUTH, D.E.: Big Omicron and Big Omega and Big Theta.
ACM Sigact News 8, 2, p. 18

[Kru56] KRUSKAL, J.B.: On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem.
Proc. Amer. Math. Soc. 7 (1956), p. 48–50

[Kur30] KURATOWSKI, K.: Sur le Problème des Courbes Gauches en Topologie.
Fund Math. 15 (1930), p. 217–283

[Law76] LAWLER, E.G.: Combinatorial Optimization: Networks and Matroids.
Holt, Rinehart and Winston, New York-San Francisco-London, 1976

[Liu68] LIU, C.L.: Introduction to Combinatorial Mathematics.
McGraw-Hill, New York, N.Y., 1968

[Lue80] LUEKER, G.S.: Some Techniques for Solving Recurrences.
ACM Computing Surveys 12, 4 (1980), p. 419–436

[MaR80] MAJSTER, M.E., REISER, A.: Efficient On-line Construction and Correction of Postition Trees.
SIAM J. Comput. 9 (1980), p. 785–807

[McC76] MCCREIGHT, E.M.: A Space-economical Suffix Tree Construction Algorithm.
J.ACM 23 (1976), p.262–272

[Mil60] MILNE-THOMSON, L.M.: The Calculus of Finite Differences.
Macmillan, London, 1960

[MiV80] MICALI, S., VAZIRANI, V.V.: An $O(\sqrt{|V|}|E|)$ Algorithm for Finding Maximum Matching in General Graphs.
Proc. 21st Ann. Symp. on FOCS (1980), p. 17–27

[MPM78] MALHOTRA, V.M., PRAMODH KUMAR, M., MAHESHWARI, S.: An $O(|V|^3)$ Algorithm for Finding Maximum Flows in Networks.
Computer Science Program, Indian Institute of Technology, Kanpur 208016, India, 1978

[PFY73] PRATT, V.R., YAO, F.F.: On Lower Bounds for Computing the i-th Largest Element.
Proc. 14th Ann. IEEE SWAT (1973), Iowa City, Iowa, p. 70–81

[Rei77] REINGOLD, E.M., NIEVERGELT, J., DEO, N.: Combinatorial Algorithms: Theory and Practice.
Prentice-Hall, Englewood Cliffs, N.J., 1977

[Rei78] REISER, A.: A Linear Selection Algorithm for Sets of Elements with Weights.
Information Processing Letters 7 (1978), p. 159–162

[Rio58] RIORDAN, J.: An Introduction to Combinatorial Analysis.
J. Wiley, New York, N.Y., 1958

[Rio68] RIORDAN, J.: Combinatorial Identities.
J. Wiley, New York, N.Y., 1968

[Sch78] SCHNORR, C.P.: An Algorithm for Transitive Closure with Linear Expected Time.
SIAM J. on Comput. 7 (1978), p. 127–133

[Shi78] SHILOACH, Y.: An $O(nl\log^2 l)$ Maximum-flow Algorithm.
STAN-CS-78-702, Department of Computer Science, Stanford University, Stanford, 1978

[Sle80] SLEATOR, D.D.K.: An $O(nm\log n)$ Algorithm for Maximum Network Flow.
Department of Computer Science Report No. STAN-CS-80-831, Stanford University, Stanford, Dec. 1980

[SPP76] SCHÖNHAGE, A., PATERSON, M., PIPPENGER, N.: Finding the Median.
JCSS 13 (1976), p. 184–199

[Tar75] TARJAN, R.E.: Efficiency of a Good But Not Linear Set Union Algorithm.
J.ACM 22 (1975), p. 215–225

[Tar77] TARJAN, R.E.: Reference Machines Require Non-Linear Time to Maintain Disjoint Sets.
Proc. 9th ACM STOC (1977), p. 18–29

[Tar79] TARJAN, R.E.: Applications of Path Compression on Balanced Trees.
J.ACM 26 (1979), p. 690–715

[Tut47] TUTTE, W.T.: The Factorization of Linear Graphs.
J. London Math. Soc. 22 (1947), p. 107–111

[Vui78] VUILLEMIN, J.: A Data Structure for Manipulating Priority Queues.
Comm. ACM 21 (1978), p. 309–315

[War62] WARSHALL, S.: A Theorem on Boolean Matrices.
JACM 9 (1962), p. 11–12

[Wei73] WEINER, P.: Linear Pattern Matching Algorithms.
Proc. IEEE 14th Ann. SWAT (1973), p. 1–11

[YAR77] YAO, A.C., AVIS, D.M., RIVEST, R.L.: An $\Omega(n^2 \log n)$ Lower Bound to the Shortest Paths Problem.
Proc. 9th ACM STOC (1977), p. 11–17

[Zah73] ZADEH, N.: More Pathological Examples for Network Flow Problems.
Math. Programming 5 (1973), p. 217–224

END

DATE
FILMED

10-82

DTIC